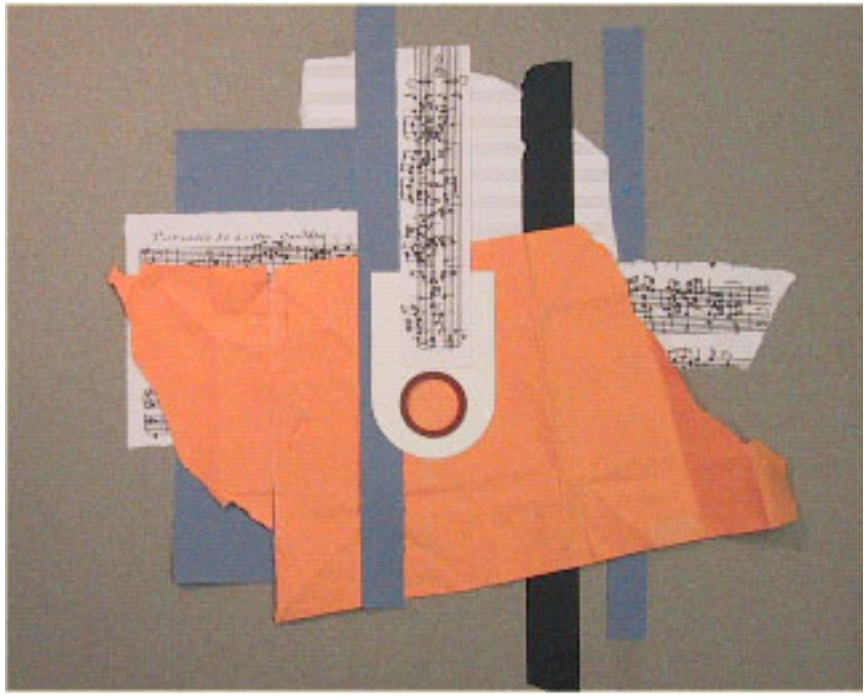
ircam
Centre Pompidou

Software
Documentation



OpenMusic User's Manual

by Carlos AGON, Gérard ASSAYAG, Jaccobo BABONI, Jean BRESSON, Karim HADDAD, Matthew LIMA, and Mikhail MALT

OpenMusic User's Manual

Edited by Karim Haddad

Published Date: 2004/02/03 14:26:48

This manual can not be copied, in whole or part, without the written consent of IRCAM.
This manual was produced under the editorial responsibility of Karim Haddad, Marketing and Communication Dept. , IRCAM.
OpenMusic was conceived and programmed by Gérard Assayag and Carlos Agon.
Patchwork was conceived and programmed by Mikael Laurson, Camilo Rueda, Jacques Duthen, Gérard Assayag and Carlos Agon.
This documentation corresponds to version 4.6 or higher of OpenMusic.
OpenMusic™ is a trademark of IRCAM.
Patchwork™ is a trademark of IRCAM.
MidiShare™ is a trademark of GRAME.
Macintosh Common Lisp™ is a trademark of DIGITOOL, Inc.
Apple Macintosh™ is a trademark of Apple Computer, Inc.

Acknowledgments:
Markus Buser.

Copyright © 2003 IRCAM. All rights reserved.

The use of this software and its documentation is restricted to members of the IRCAM software users' group. For any supplementary information, contact:

Marketing and Communication Dept.,
IRCAM
1 place Igor-Stravinsky
F-75004,Paris
FRANCE
Phone:(33) (1) 44 78 49 62
Fax: (33) (1) 42 77 29 47
Email: palumbo@ircam.fr

Send comments or suggestions to the editor:

Karim Haddad
Marketing and Communication Dept.,
IRCAM
1 place Igor-Stravinsky
F-75004,Paris
FRANCE
Phone:(33) (1) 44 78 16 05

Email: haddad@ircam.fr

For more information:
http://www.forumnet.ircam.fr

Table of Contents

Foreword ..... **i**

1. How To Use This Manual **1**

Organization of the Manual 1

What is assumed 1

Notation Conventions 1

I. Getting Started **1**

1. Installing OM 3

The distribution CD-ROM 3

Installing Midishare® 3

Starting OM 7

2. Quick Tour 9

What is OpenMusic? 9

Object-oriented programming with OM 9

The Workspace 9

The Listener Window 9

The Workspace Window 10

Folders 11

The Packages Folder 11

The Globals Folder 11

Maquettes 11

Patches 11

Creating a new patch 12

Adding icons to a patch 14

Passing data to inputs 15

Invoking the built-in documentation 15

Passing data to inputs, continued 16

Evaluating a patch 17

Factory icons 19

Classes and factories 19

Adding a factory 19

Editing music objects 19

Giving input to factories 20

That's it! 21

II. OM Concepts **23**

1. Macintosh Common Lisp 25

The Listener Window 25

2. The Workspace 27

Managing Workspaces 27

The Menu Bar 27

The File menu 28

The Edit menu 28

Presentation 28

Objects in the Workspace 28

Manipulating Items in the Workspace 29

Drag & Drop 29

Import and Export 29

Contextual menus 29

3. Patches 31

The in-line documentation....31

Patches in the Workspace....33
Patch structure....33
Inputs and outputs....34
Objects within the patch....34
Making connections within the patch....35
Evaluating patches....36
Variables....37
Manipulating icons in the Patch Window....37
Abstractions....37
Contextual menus....38
Object states....38
4. Packages....41
Managing Classes....42
Inheritance....42
Slots....43
Initialization methods....43
Managing Functions....44
Method definitions....44
5. Globals....45
6. Data Types....47
7. Rhythm Trees....49
The structure....49
Some Examples....50
8. Functions....53
Functions and methods....53
Function definitions....53
Dead methods....53
Creating new functions....54
Creating the function name and appearance....54
Defining the first method....55
Adding additional methods....56
9. Classes, Instances, and Factories....59
Class Definitions....59
Slots....59
Factories and Instances....59
Ins and Outs....60
Inheritance....60
Variables....61
Creating a new class....62
Setting the inheritance....62
Adding slots to classes....63
Changing the initialization method for a class....65
The get-set-slot construction....66
10. Maquettes....69
The structure of a **Maquette**....69
Objects in the **Maquette** frame....70
Acceptable types....70
Adding objects to the **Maquette**....70
The **Maquette** frame....70
The General Palette....71

Contextual menus 72

The Metric Ruler 72
Markers 73
The **Temporalbox** object 73
The *self* outputs 74
Maquettes in patches 74
11. Midi and OpenMusic 75
Introduction to MIDI in OpenMusic 75
Basic concepts 75
MIDI messages 75
About MIDI events 76
Note events 76
Control Change events 76
Program Change Events 76
`Pitchbend` events 77
Using MIDI in OpenMusic 77
Sending MIDI events with simple functions 77
Selection tools 79
The Settings Controller 80
MIDI Files 81
The **MidiFile** box 81
Saving OpenMusic objects in MIDI files 82
MidiFile informations 83
MIDI Events 85
The **MidiEvent** box 85
Processing **MidiEvent**s 86
MIDI sequences 89
The **EventMidi-seq** box 90
Using Midi sequences for musical objects operations 92
MIDI Continuous Controllers 93
The **MidiControl** box 93
Continuous controllers values 95
Creating **MidiControl**s 95
Tempo and Measures 97
The **Tempo-Map** box 97
Improving quantification 98
MidiShare MIDI events table 100
12. Preferences 103
The General Preferences 103
Special file writer and Message Error handle buttons 103
Comment Style button 103
Comment Color button 103
User Name field 103
Screamer Valuation buttons 103
The Midishare Preferences 103
The Notation Preferences 104
Enharmonic spellings 104
Music Font Size field 105
Delta Chords field 105
Colors buttons 105
Staff System menu 105

Quantify button 105

Dynamic Table 105
The Csound Preferences 105
13. Libraries 107
OM 4.6 Glossary of Terms 109

Foreword

OpenMusic is a sea in which a gnat may drink or an elephant may bathe.

ancient Indian saying

Welcome to OpenMusic! OpenMusic, or OM, is a *graphic user interface* for the popular *LISP programming language*. OpenMusic is just that: open. It was not created for any one task but rather as a dynamic environment which you gradually adapt to your work. It is thus impossible to completely document the functionality of the software, since you, the user, will ultimately decide what you'll do with it. Besides the graphic interface to Lisp, OM implements a great set of music objects, operators, and music-specific editors.

This flexibility is the appeal of OM: Composers can use OM to refine aspects of their musical language requiring computation without resorting to cumbersome code or calculating by hand; Musicologists can use OM to perform analytical computations on musical material (in the form of notation or MIDI files); Functional programming lovers can enjoy one of the best graphic programming interfaces developed to date, independent of its musical features.

OM was and continues to be written by Carlos Agon and Gérard Assayag at Ircam. It follows on the heels of PatchWork, a piece of music software previously developed at Ircam by M. Laurson, C. Rueda and J. Duthen, with contributions by C. Agon and G. Assayag.

This documentation has been significantly improved for the new release of the software. There is now a comprehensive glossary containing definitions to key terms you might not be familiar with. The Reference section of this Manual has been revised and completed, and now includes both classes and their graphic editors. There is also an index to the entire Manual, including the Tutorials, which have been reorganized and revised. We will be producing an html version of this document. The Function Reference will be searchable by keyword, and the entire Manual will be cross-referenced with numerous hyperlinks. In writing this manual, we have erred on the side of caution, imagining our least experienced reader to be a composer with lots of motivation and curiosity but almost no computer expertise. Much of this manual will therefore seem superfluous to our more experienced users; please forgive us. Get your bearings by reading the next section, which will guide you to the material most appropriate to your skill level.

This documentation is a collective work with numerous contributions by Mikhail Malt, Karim Haddad, Jacopo Baboni. Matthew Lima has reorganized all the materials for this new edition, written several new parts and fleshed out others, and last but not least, transformed the text from its original "Franglish" incarnation to normal English.

Questions and comments about the software and the documentation should be addressed to `<openmusic@ircam.fr>`, or `<forum-support@ircam.fr>` if you are a member of the Ircam User Forum.

Have fun!

Chapter 1. How To Use This Manual

Organization of the Manual

This manual is divided into four main parts:

1. Getting Started - how to install and launch the software, and a quick tour intended to demonstrate the basic notions of the software.
2. The OpenMusic Concepts - The meat of the Manual, being a systematic description of the elements of the OM environment. I've put them in what seems the most logical order, but there is not necessarily any continuity between the sections. These chapters are comprehensive but not progressive.
3. The Tutorials - the counterpart to the Concepts. Here, a series of tasks are presented, in increasing order of complexity. Unlike the Concepts section, only what is directly relevant to the task at hand is discussed. These chapters are progressive but not comprehensive.
4. Reference - a series of indexes. All the *functions* and *classes* are documented on their own pages. There follows a glossary and an index.

What is assumed

As mentioned before, we've assumed a lot of motivation but little expertise on the part of the reader. We do assume you know how files and folders are heirarchically organized on your computer, and how to open, close, and move them around. If this describes you, but only just, you probably should start at the beginning of the Getting Started section, install OM, set it up, then take Quick Tour from start to finish. Then, start the first of the General Tutorials. Work your way through the tutorials, refering to the Reference and the Concepts as you feel necessary.

Users very comfortable with computers but unfamiliar with object-oriented programming should should take the Quick Tour and then go to the General Tutorials, skimming or skipping the first few until things start looking interesting.

Users with a bit of OM experience will be primary interested in the newly rewritten Concepts section and the all-new entries on the graphic editors in the Reference section.

Notation Conventions

As in all computer manuals, we've adopted certain notation conventions throughout the documentaton for clarity. The following typographical rules apply:

When we talk about keys on the computer keyboard, they are represented **like this**.

Examples: **a**, **r**, space

Combinations of keystrokes are sometimes required, meaning that one key is held while another is pressed or while the mouse is clicked:

Examples: option-**i**, command-**a**, option-→.

The possible keys to hold down are ctrl, option, and command (also known as the apple key). The one exception is shift with letter keys, where I've chosed simply to indicate the uppercase letter. The cursor movement keys are represented by the arrows ← → ↓ ↑.

When talking about selecing items from the pulldown menus, they are indicated in the order you must select them from the top level. For example, selecting the item "New Patch" from the "New" submenu in the "File" pull-down menu at the top of the screen is represented: File⟶New⟶New Patch.

Example: Functions⟶music⟶score⟶OMQUANTIFY

Names of functions are never capitalized and are always printed `like this`. Names of classes are always capitalized and printed **like this**. Data types are not normally capitalized (except when a class is a data type) and are printed like this.

Examples: `first`, **Poly**, t, **Note**, `list`, integer.

All set? Then let's go!

I. Getting Started

Chapter 1. Installing OM

The distribution CD-ROM

OpenMusic is distributed bi-annually on CD-ROM by the IRCAM. By the time you read this, OpenMusic will exist in incarnations for Mac OS 9, Mac OS X, and Linux. How you install depends on which configuration you are running the software on, and will probably evolve quickly over time. For this reason, we've not included installation instructions in this manual. For the latest information, please visit the OpenMusic website at http://www.ircam.fr/equipes/repmus/OpenMusic/.

The system requirements if you are using a Macintosh are as follows:

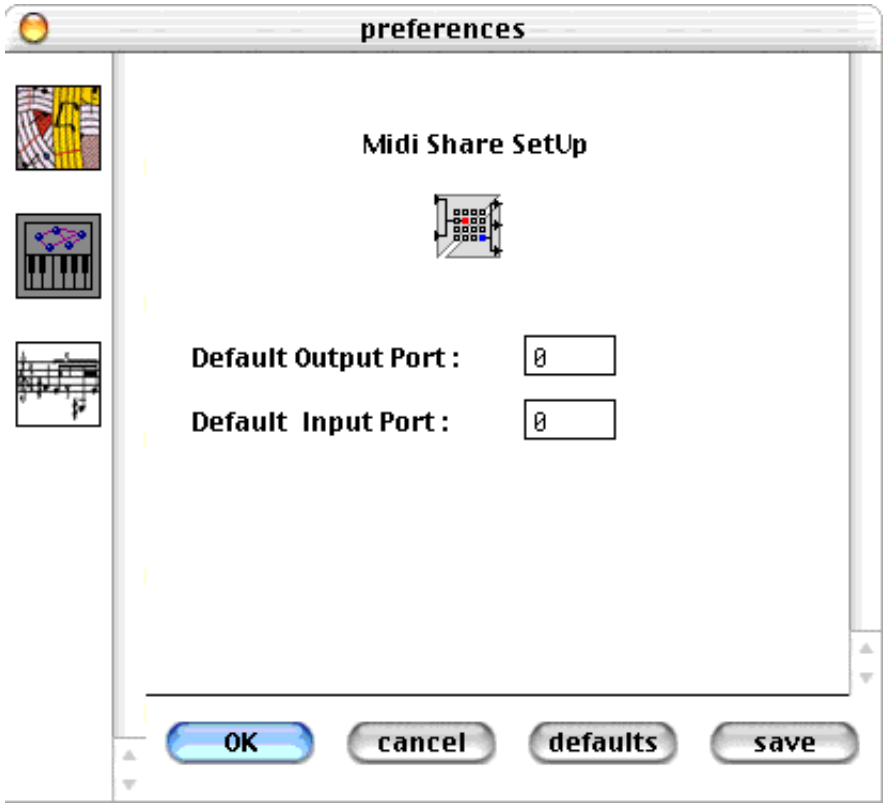
- Power Macintosh G3, G4 or G5
- System software 9.x or 10.x
- In System 9, 15 MB of RAM assigned to OpenMusic itself. (RAM is allocated automatically in system X) 64MB of installed RAM is recommended as the minimum for your machine.
- A CD-ROM drive.
- You will need to use Midishare (see below) if you want to connect OpenMusic to external MIDI instruments.

Installing Midishare®

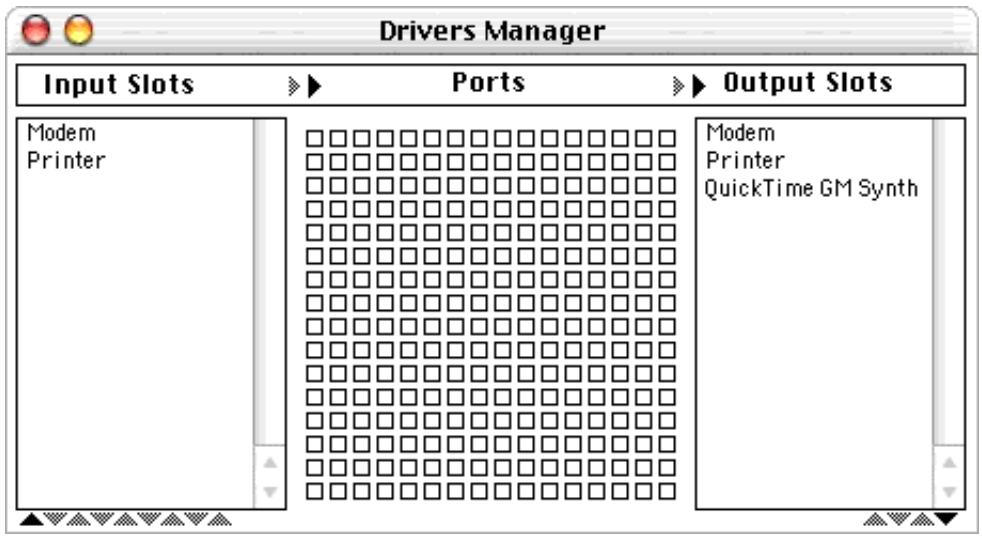
Important: As of this printing, the Midishare® software that OM uses to communicate with *MIDI* devices is also between versions. There are both OS 9 and OS 10 versions. Both included on the distribution CD-ROM. You should install whichever version is apprpriate for your computer following the instructions included with the software.

Midishare can control most kinds of MIDI equipment through a selection of *driver* software. Drivers are included for Apple's built-in QuickTime synthesizer and MIDI equipment connected to your computer and managed by Opcode's OMS software. Once the software is correctly installed and the appropriate drivers are in the appropriate places, (again, see the Midishare documentation) Midishare operates identically as far as configuration is concerned. Install Midishare before continuing.

Midishare provides 256 possible communication ports for making connections between components. We will set OM to send and recieve on port 0, the first one. Go to the preferences (File⟶Preferences) panel and click the Midishare icon (featuring the little keyboard in the lower half of the icon:)



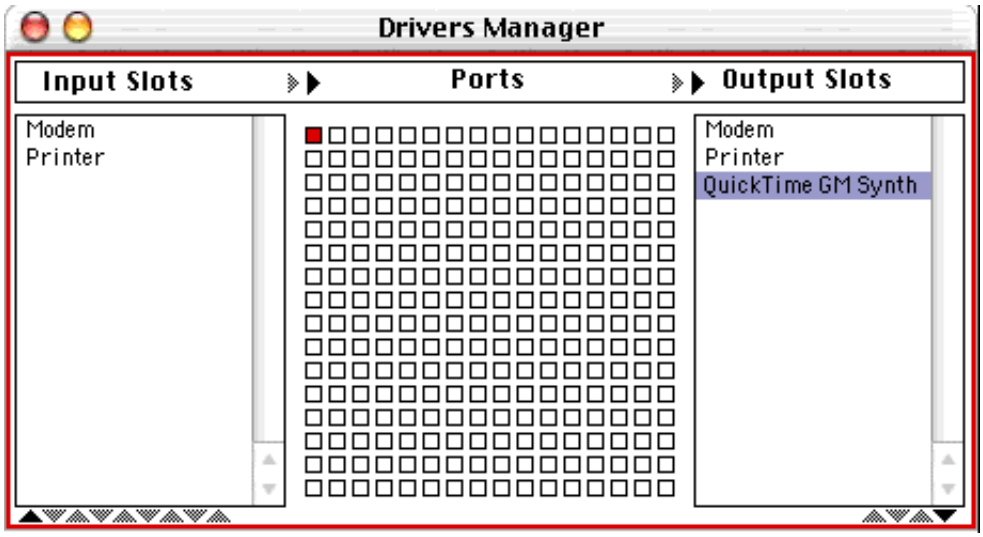
Click in each box and set the input and output ports to zero. Then click the Midishare icon in the center of the panel to open the Midishare Drivers Manager. The following window opens:



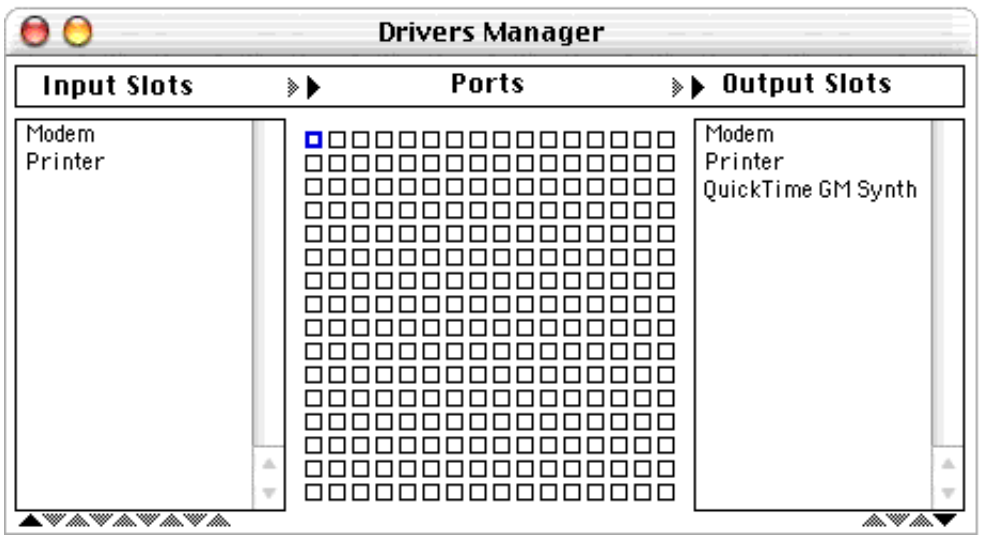
What is listed in the columns on the left and right may vary depending on what's installed on your system. If you have installed the QuickTime driver, you will see 'QuickTime GM Synth' among the output slots. If you've installed the OMS driver you'll see 'IAC Bus #1' or something like it in both the input and the output slots. If you haven't you should install the QuickTime driver as it will allow you to play objects through your computer's built-in hardware instead of depending on your MIDI rig.

Each of the boxes in the middle represents a port. In order to assign a driver to a port, one must select the driver and then choose a port (or vice-versa). Click once on the 'QuickTime

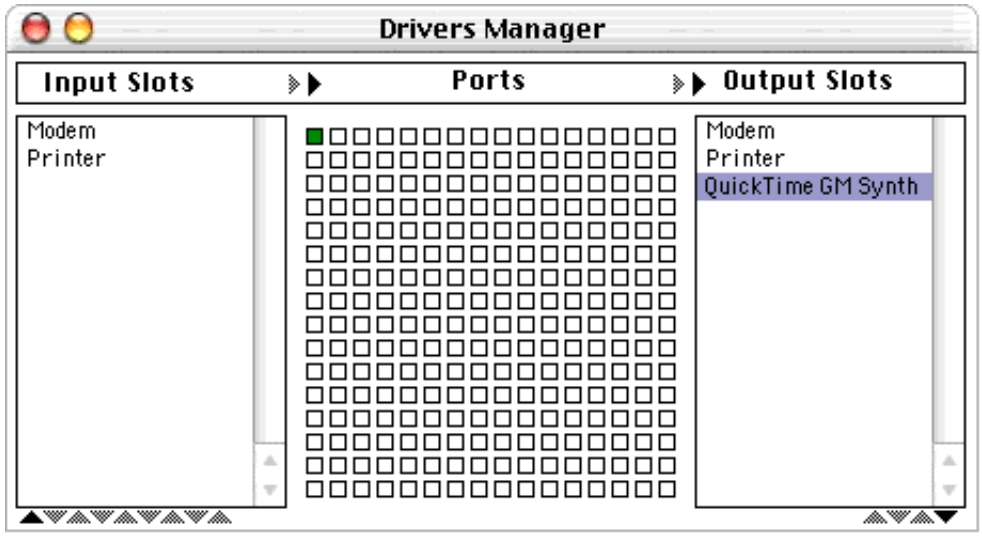
GM Synth' on the right. Click the box in the upper left of the grid. It will turn red. That means it's been assigned:



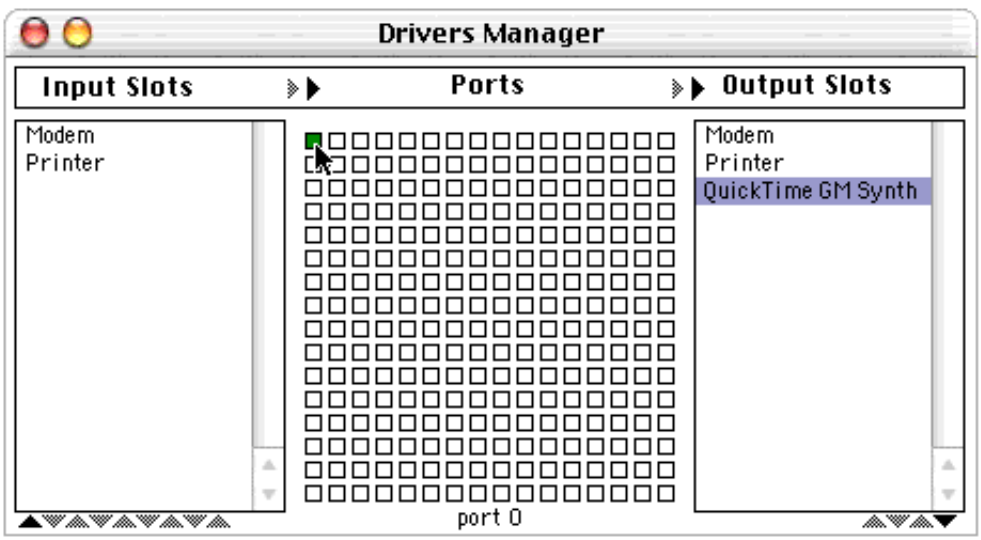
To test this, close the Driver Manager and reopen it from the Preferences:



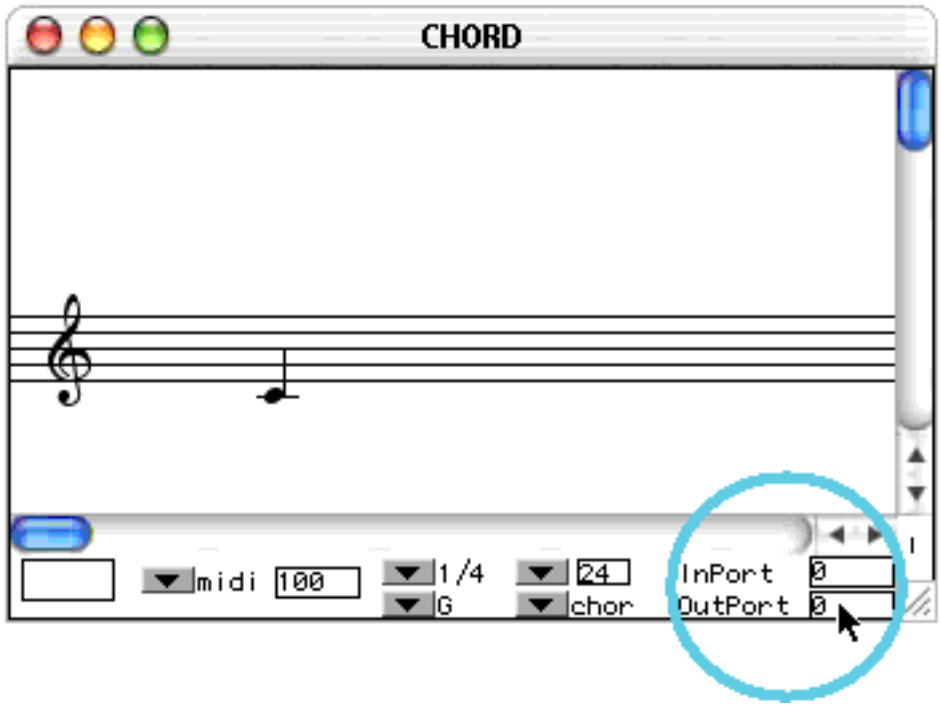
You'll notice that port zero is now outlined in blue, which indicates it's been assigned. To check what ports a particular driver has been assigned to, click its name in the Input or Output columns. The ports to which it has been assigned will turn green. Try it with QuickTime GM Synth:



Moving the mouse cursor over the green-lit port will reveal its number on the bottom of the panel below the grid:

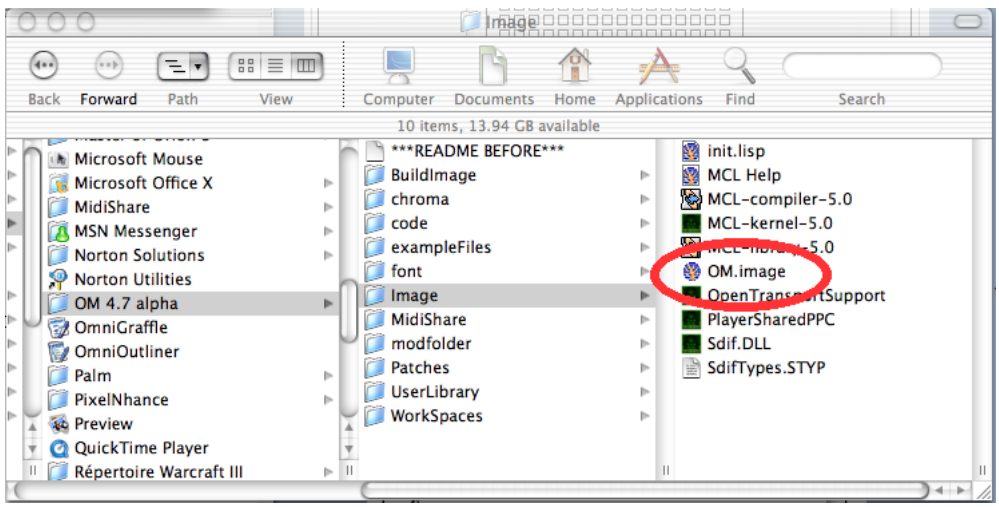


That's it! Close the panel and return to the Preferences. If you use a setup where more than one port is used, remember that OM musical objects have their own Input/Output port designations which override the default assignment and which you will have to set if you don't want the data to go out over the default port:



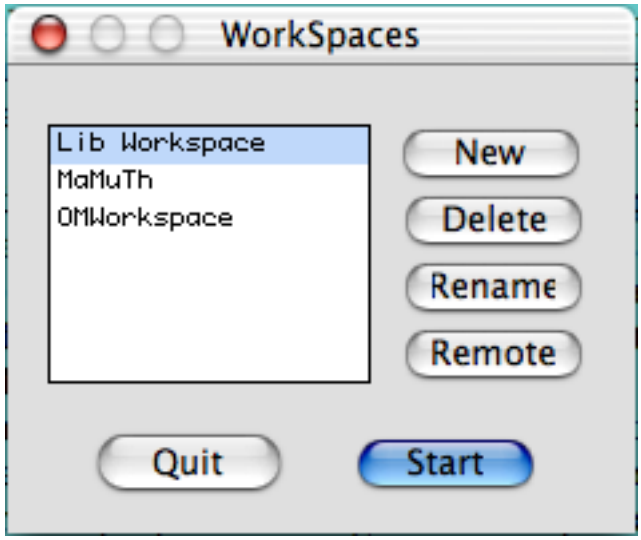
Starting OM

In both system 9 and 10, OpenMusic resides in a folder called `OM 4.x` somewhere on your hard drive (depending on where you put it at installation time). In that folder are a collection of other folders with names like `BuildImage`, `chroma`, `code`, etc.



We're interested in the folder called `Image`, which contains the OpenMusic application program, somewhat unintuitively titled `OM.image`. Double click this file to start OpenMusic.

After the title screen, you'll be asked to choose a Workspace:



The Workspace is the top level of the OM environment. Everything you do happens in the workspace. Multiple workspaces can be maintained so that several users can use the same computer. If you go back to the OpenMusic folder where you located the `Image` folder, you'll notice a folder called `Workspaces`. Each subfolder within this one is a Workspace you can choose to work in at the menu above.

> **Important:** While the folders in the `Workspaces` folder represent your Workspaces, they are managed by OM and the contents should not be modified directly. The one exception is in the event of a total OM apocalypse, where your Workspace becomes unusable, in which case you can reconstitute a completely virgin Workspace by deleting the entire contents of the `Workspaces` folder and restarting OM.

I recommend doing two things in this folder. Firstly, make a copy of the folder `OM Workspace` and call it "Empty" or "Blank" or something- just keep this copy around in case you want access to a pristine Workspace without reinstalling from the CD-ROM. Secondly, make a back-up copy of whatever Workspace you use. Command-**d** makes a duplicate of a folder and adds 'copy' to the end of the filename. At the end of each session, throw out your old copy and create a new one. This way if anything goes wrong during your session you can throw out your Workspace and make a new one from the back-up. This is important because unfortunately **there is no Undo function in OpenMusic**.

Chapter 2. Quick Tour

Note: Many terms used here are defined in the glossary to save space.

What is OpenMusic?

OpenMusic, or OM, is a visual programming environment for the *LISP programming language*, fine-tuned for, but by no means restricted to, the processing of musical objects. It is the descendant of the venerable Patchwork software. Patchwork was also a musical programming language, but had no graphic interface- commands were entered and results obtained as text. OpenMusic adds a *graphic interface* which makes it much easier to work with.

OpenMusic is written in a programming language called LISP. LISP is simply a standard syntax for writing code; many different programs exist for writing and compiling LISP code. OpenMusic was written using *Digitool Macintosh Common Lisp*, and includes a graphic interface for programming in LISP as well as CLOS (the Common Lisp Object System). This makes OpenMusic a graphic interface for an object-oriented programming environment.

Object-oriented programming with OM

All the elements you will manipulate in OM are represented by icons that you move around and interconnect graphically with a combination of keyboard and mouse commands. The types of musical objects you can manipulate include Musical Notation (both measured and non-measured), Audio (AIFF) files, and *MIDI* files. Eventually, you will be able to create new objects to fill your particular needs.

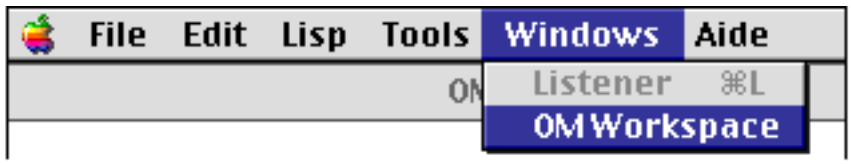
The Workspace

Everything you do in OM happens in the Workspace. Like the desktop on your Macintosh, the Workspace is the highest organizational level in OM. When you open OM, you will see two windows; the Listener and the Workspace.

The Listener Window

When you run OM, you're actually using the Digitool Macintosh Common Lisp programming tool, as mentioned above. OM runs 'on top of' the Digitool Environment, if you will. The Listener is your direct connection to the Digitool Macintosh Common program.

Click on the listener. You'll notice the menu at the top of the screen changes to include items like "Lisp" and "Tools." Return to the OMWorkspace window by clicking on it or selecting it from the Windows menu.



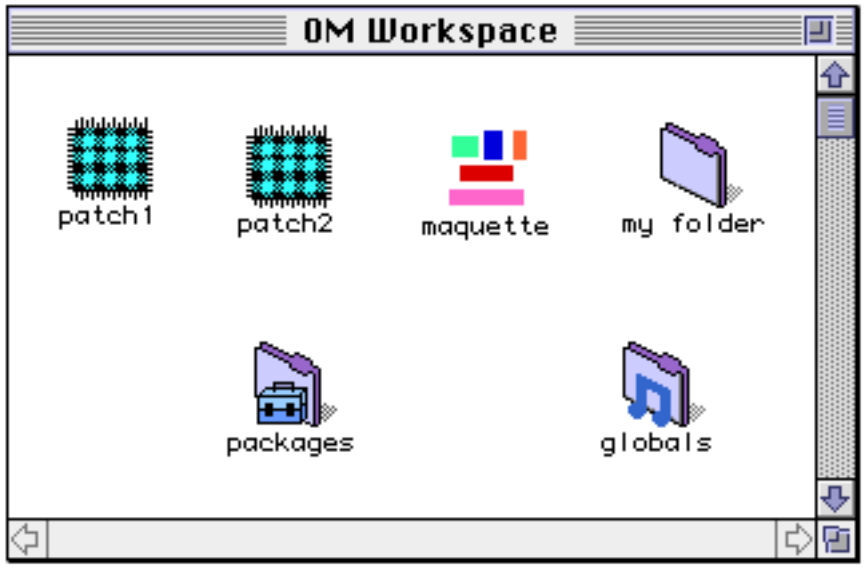
...and you'll see the menu bar at the top of the screen change to include elements like "Presentation." This menu bar will always change depending on where you are in the Workspace. Digitool MCL prints its output to a window called the Listener. This output may be the result of a calculation, an acknowledgement that a task is finished, or an error message if you screw something up or tell MCL to do something it doesn't like, which might look like this:

During your sessions you will undoubtedly encounter many such errors. When it encounters an error, MCL stops whatever it's doing and waits for you to tell it how to handle the error. When this happens, click the Listener and select Lisp⟶Abort or hit Command-**.** (Command-**period**) If you continue other tasks before you abort the error, they may not run correctly.

Now let's look at the objects in the Workspace Window.

The Workspace Window

The Workspace window looks like and behaves similarly to the Mac OS 9 Finder. You'll probably see a larger version of something like this:



Objects in the Workspace can be dragged from one place to another. In fact, you program in OM by dragging objects to patch windows. There are three types of objects in the Workspace: folders, patches, and maquettes. Objects in OpenMusic are persistent, which means they stick around as long as you don't specifically delete them. The Workspace appears as you left it at the end of your last session. To delete an object, click on it and select Edit⟶Clear. By careful here; there's no Undo feature in OM.

Folders

Folders function exactly as they do in the Mac Finder. Create them with Command-**n** or the menu item File⟶New⟶New Folder There are two special folders in the Workspace which cannot be deleted: Packages and globals.

The Packages Folder

The packages folder contains all the 'stuff' OM is made of. A package is a set of objects. Double clicking the packages folder displays the available packages. You'll create your own packages later. Leave this alone for now.

The Globals Folder

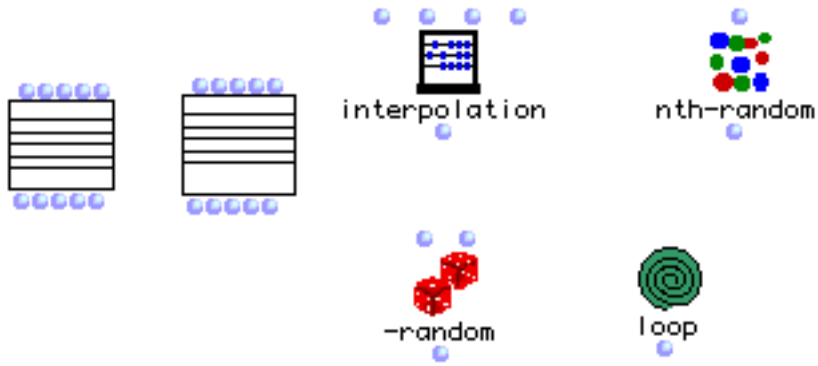
The globals folder stores variables (i.e. class instances) that you want to share among patches. Don't worry about this for now, either.

Maquettes

Maquettes are a type of object which allows other objects to be connected in temporal relationships. They're not covered in the Quick Tour. See the Tutorials section instead.

Patches

The *patch* is the basic unit of programming in OM. Generally speaking, a patch can consist of three types of object linked up to create a flowchart representing a task. They are the basic building blocks of object-oriented programming, and they are represented by icons on the screen. These objects may also be called boxes or modules- the terms are synonymous. The first two types of object are *functions* and *factories*. Functions are, as their name suggests, boxes which perform some kind of calculation on their input and output the result. Factories are boxes which produce a certain type of object representing some kind of data. Here are two (empty) factory icons and a few function icons:



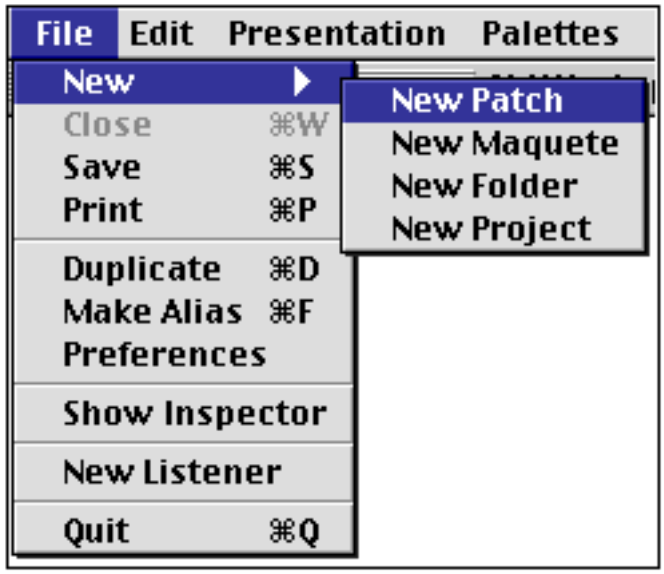
Each factory icon knows how to produce only one type of data structure. The blueprint for this data structure is called a *class*. The factory takes inputs and produces an object based on the model of a particular class. Classes in OM include notes, chords, audio files, numbers, and lists- anything we use as raw material.

The third type of object is another patch- once you've written a patch, you can use it as a subpatch in other patches. This modular approach saves you lots of time and is one of the most powerful aspects of working with OM. Patches can even be put within themselves, creating what's called a *recursive patch*.

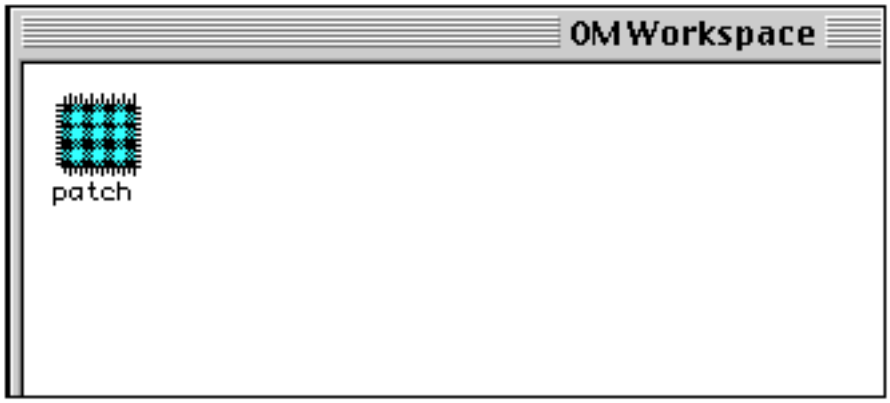
We're now ready to put together a very simple patch, in which we'll add 700 to 6000, and turn this number into a pitch using a **Note** factory.

Creating a new patch

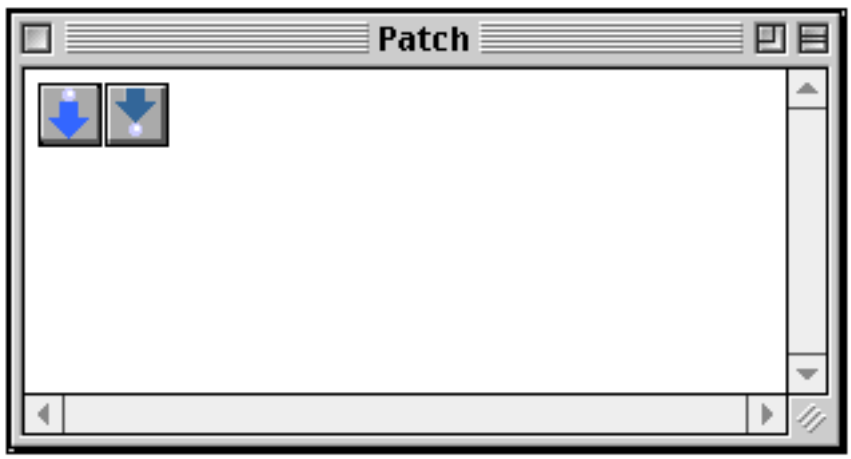
In the workspace window, choose File⟶New⟶New Patch.



A new icon appears in the Workspace, possibly on top of an already-present icon, in which case you should just drag it to a clear part of the Workspace:



Double-click the patch icon to open the patch window:

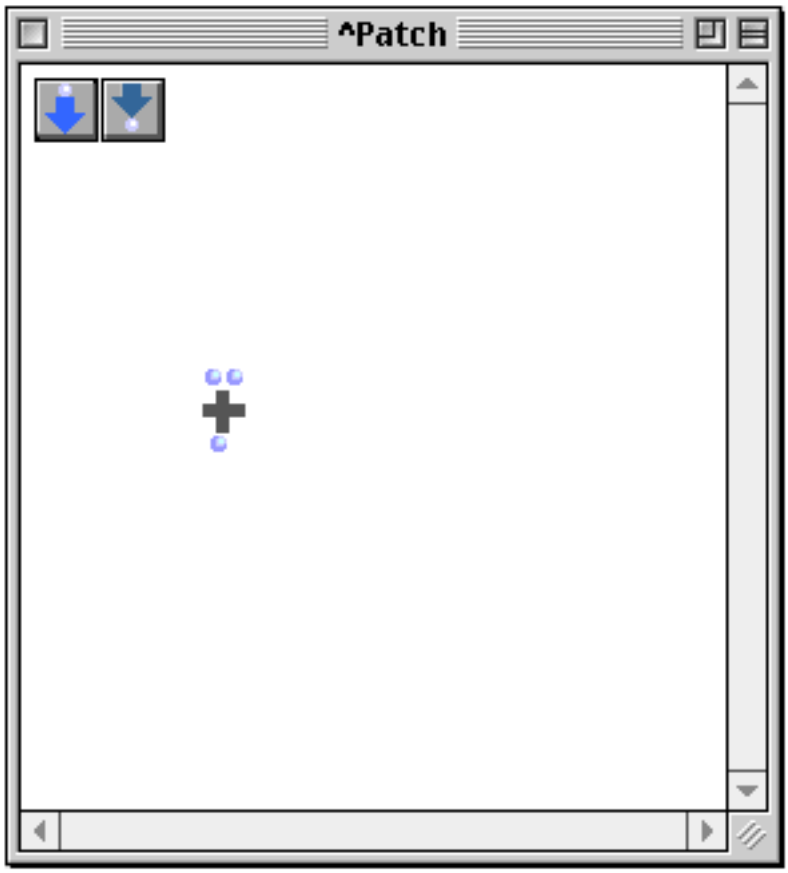


The two buttons in the upper left are for adding inputs and outputs to the patch as a whole

to permit linking it with other patches; don't worry about those for now.

Adding icons to a patch

First, we'll add the function that will add the two numbers. From the pull-down menus, select Functions⟶Kernel⟶Arithmetic⟶om+. The cursor changes to indicate you are placing a function. Click somewhere in the patch window and you'll see the function icon:



There are actually three ways to place an object in the patch window. The first is by menu selection, as you've just done. The second is to type the name of the function directly into the patch window. Command-click anywhere in the patch window and type the name of the function (in this case `om+`) and hit enter. The function icon pops up. You can delete this second icon; we won't need it. The third way is to drag the icon of the function or factory you are placing from another patch window, or from the packages folder in the Workspace. The packages folder is explained in its own chapter.

Check out the icon for `om+`. You'll notice two blue points above the function- these are the inputs. Any blue dot above an icon is an input. Obviously enough, the blue dot below the function is the output. Most functions have one, but a few have more than one.

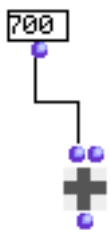
Now we need to tell `om+` what to add. Command-click somewhere above the function in the patch window. Instead of typing a function name, type the number 700 and hit enter.

> **Return Vs. Enter:** Remember that values are confirmed in OM with the enter key, not the return key. Use return only if you want a carriage return, in a comment box, for example.

If you had typed text, OM would have looked for a function or class corresponding to the name. Since you typed a number, a box representing the number pops up. If you messed that up you can double-click the box and adjust the quantity.

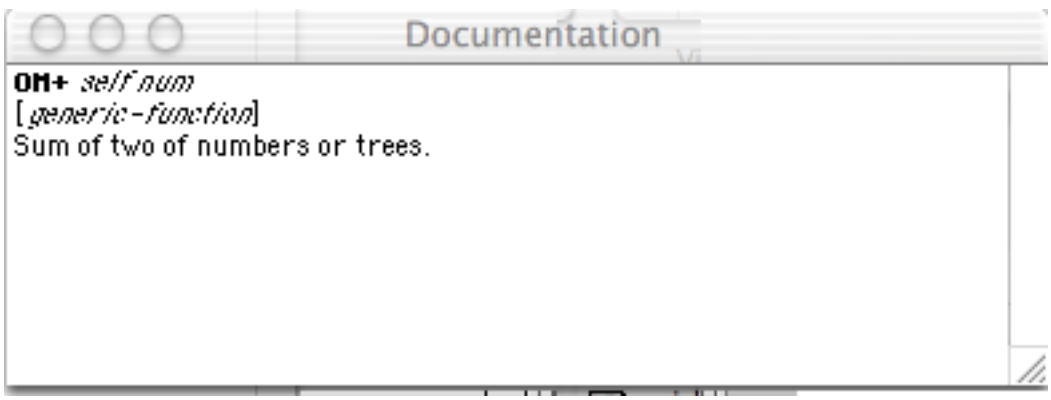
Passing data to inputs

You'll notice the 700 box only has an output. That's because all it will do is output whatever quantity it is set to. Click the output and drag a connection to the first input of the `om+` icon. When you do, a bubble pops up with a bit of info about the input. When you release the mouse button, a connection is established:



Invoking the built-in documentation

While we're here, you should know that OpenMusic has some on-line help. You can access the documentation for a particular object by selecting it with the mouse and hitting 'd'. Try it now with the `om+` icon. This pops up:

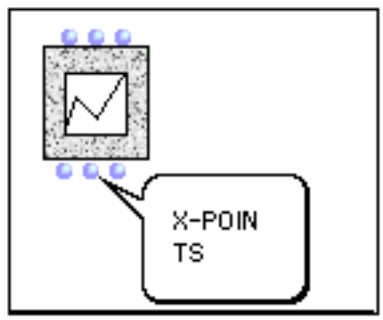


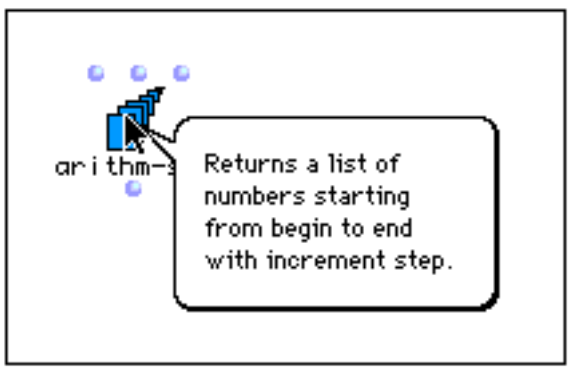
You see the function name, the names of the two inputs in italics just after it, the type of object, and some information about what it does.

The built-in documentation: This built-in help can be a bit dicey. Eventually, it will be brought up to date in the same way as this manual. For now, it's a little like playing the lottery. You may get just the names of the inputs. You may get the information you need. You may get the information you need but in French or another language. Or you may get nothing at all. Always check this manual if you don't find what you want.

Remember the little help bubble that popped when you were connecting the 700 to `om+`? You can get that help bubble to pop up at any time by holding command and holding the mouse button down on any object, any input, or any output.

When you start connecting boxes in OpenMusic you often need to know the types of input a module can receive. Command-click on an input will issue a balloon-style help. The bubble that pops up will tell you about the input or output or the function itself:

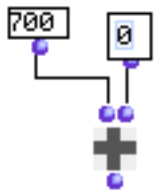




Passing data to inputs, continued

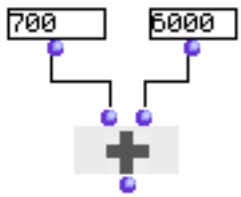
Move the mouse over the right-hand (unconnected) input. You'll see the number 0. This 0 represents the default setting of that input. When nothing is connected to an input, you can see the default setting by moving the mouse over it.

You can save time by creating the data and the connection at the same time by shift-clicking on an input. Keeping the mouse over the right-hand input, shift-click it. A box comes up with the default value in it, connected to the input and ready for you to edit the value:

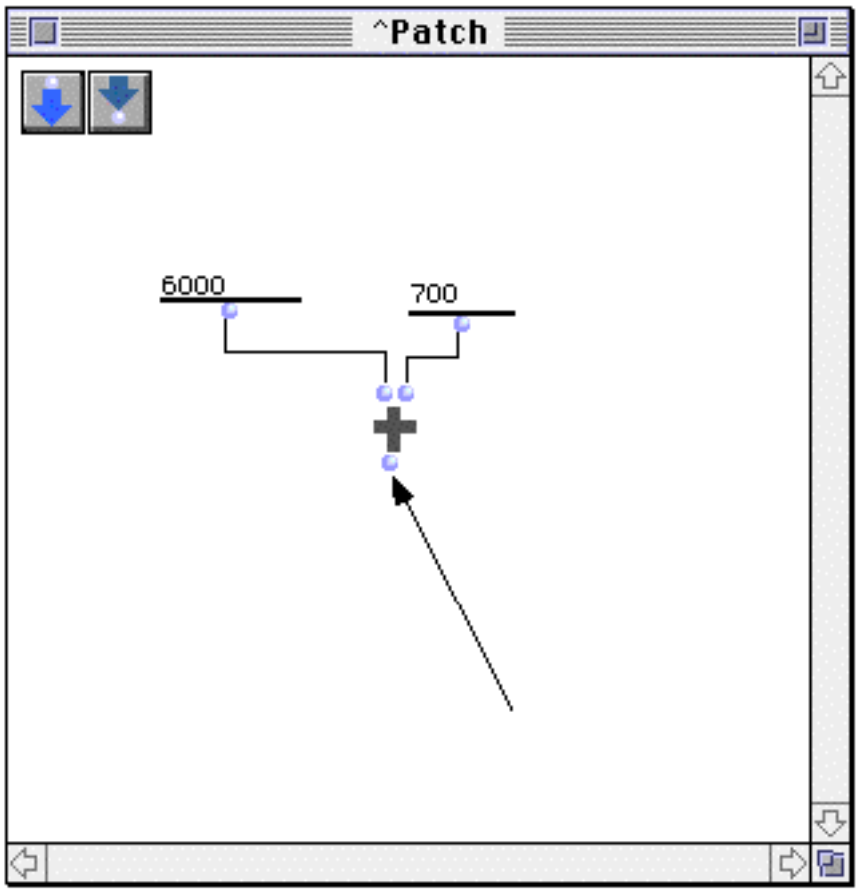


Type 6000 in and hit enter.

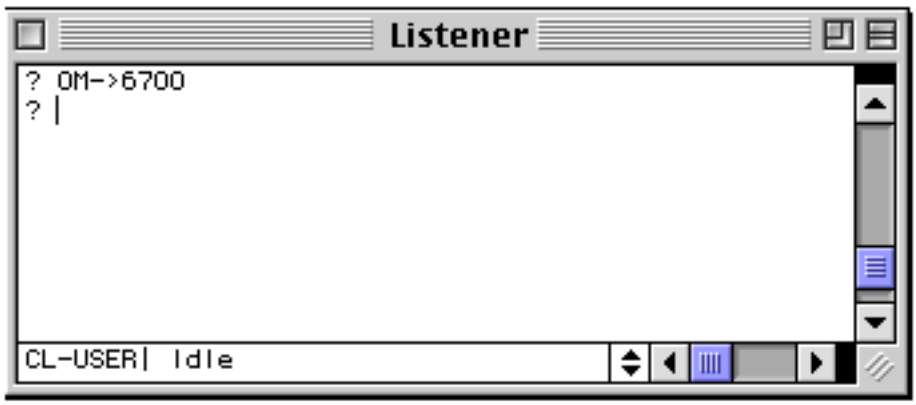
You'll notice that the box does not automatically adjust to display the full value (though it does contain it.) The size of any object can be adjusted by moving the mouse pointer over the lower-right corner of the box. The mouse cursor changes to two small arrows. You can then drag the box out to the right size. Change the box so that you can see the entire number 6000.



Evaluating a patch



Now you are ready to evaluate this little patch. Option-click the output. (The blue point below the icon.) The result will appear in the listener window:



You may also just click the `om+` icon and hit **v** which has the same effect for functions with one output. (For functions with two outputs, **v** always returns the leftmost one. You'll need to `option`-click for the others)

Don’t leave errors unfixed!: Occasionally you will encounter errors while programming your patches. It is very important to fix the part of the patch that generated the error before quitting out of OM. This is important because when you open a patch OM reconstructs it from the save file. Reconstructing the patch can generate the same errors as evaluating the patch, and this can result in a patch which you simply can’t open because it generates an error every time you try to open it.

Therefore, resolve the errors BEFORE leaving OM. If you can’t resolve the error, disconnect or delete the objects producing the error. If you don’t you risk losing the contents of the patch.

Factory icons

Classes and factories

In our example we’ve added two numbers together using `om+` we will convert this number into a pitch with the aid of a factory box. As mentioned earlier, a class is a model of a data structure. A factory produces a rubber-stamp of a class. We will add a factory icon for a single note, and pass the 6700 to it as measure of its pitch.

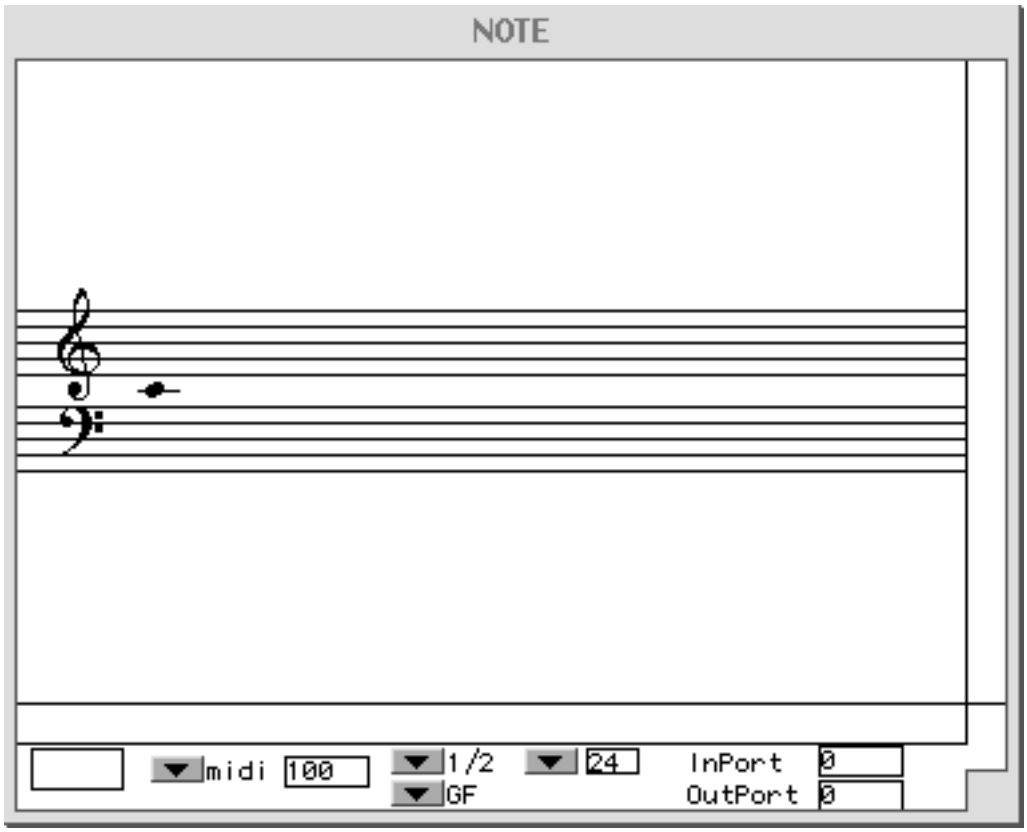
Adding a factory

From the Classes menu, select Classes⟶Music⟶Score⟶Note. Again, the cursor changes. Place the **Note** somewhere below the `om+` icon. As with functions, you could have typed the name "note" directly in by option-clicking, or dragged a **Note** factory icon from the packages folder. At any rate, you’ve just placed a note factory:

You’ll notice that the factory icon has as many outputs as inputs. This is always the case. Command-click each of the **Note** inputs in turn, starting from the left. The first, *`self`*, is common to all factory objects. *`self`* represents the object in its entirety. The other inputs have different names and depend on the factory in question. For the **Note** object they are: *`midic`*, *`vel`*, *`dur`*, and *`chan`*, and each of these inputs take some piece of data about the **Note**- its pitch (midic), volume (velocity), duration (in milliseconds) and MIDI channel. Other factories have many more inputs and outputs. These quantities are all returned separately by the corresponding outputs at the bottom of the function. The *`self`* output returns the entire **Note** object, which contains all of these values. You can verify this by holding the mouse over each input, which will display the default settings for the **Note**, then option-clicking the corresponding outputs, which will return these quantities.

Editing music objects

Midicents, or *midics*, are OM's way of representing pitch. 6000 is middle C and each semitone is equivalent to 100 midicents. (See the glossary.) Notice that the default value for the `midic` input is 6000. Double click the **Note** icon. An editor pops up:



Our note is displayed here, along with a palette of tools. All music objects have graphic editors like this, in which you can edit the object with the mouse. For now, close the editor. Now click on the **Note** icon and hit **m**. This turns on a mini-view of the contents of the **Note**. Using option and the cursor ↑/↓ keys allows you to move the view so you can see what's in there:

Giving input to factories

This miniview exists for all notation objects, too. Hit **p** to hear the **Note** (provided you've set up Midishare). Now, we want to change the pitch of the **Note**. We will do this by connecting the output of `om+` (=6700) to the `midic` input of the **Note**. When we evaluate the **Note** factory,

(hitting **v**) the new midic value of 6700 turns the **Note** into a G:

You'll also notice that the second output of the **Note** object now returns 6700. Try setting the *dur* input to 3000 to change the length of the **Note**, remembering to evaluate it before listening in order that the changes be made.

That's it!

That's the basic scheme of building patches in OM. Data is processed by functions and given to factory icons to build musical objects. The reverse is also common- taking data from a musical object, and analyzing it in some way with functions, maybe passing it back to a factory to display the results. From here you may jump to the Tutorial section or read up on the OM concepts for in-depth info about the Workspace and its components.

II. OM Concepts

The Meat of the Matter

This part of the Manual is devoted to an exhaustive description of the elements of the OM environment. They are not presented in any particular order, and can get quite technical, which may cause some confusion to the neophyte. It is recommended that users who are taking their first steps with the software read the Tutorials first.

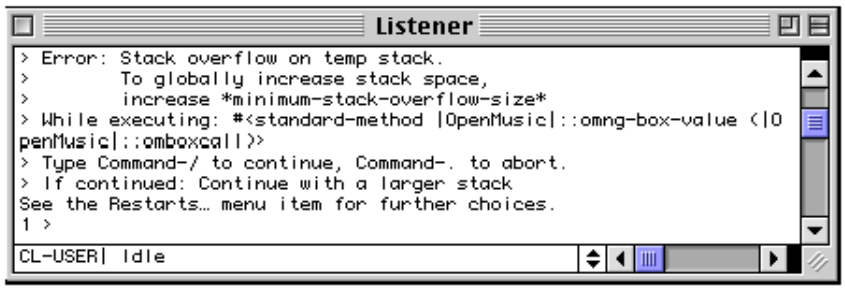
Chapter 1. Macintosh Common Lisp

OpenMusic is an interactive graphic environment for composition and MIDI processing. It is a spiritual descendant of the Patchwork software some of you may be familiar with. However, OpenMusic implements a set of radically new features that make it a second generation compositional tool. OpenMusic, or OM, is written in, and runs on top of, Digitool *Macintosh Common Lisp*. It provides a visual programming interface for Lisp programming as well as to CLOS (Common Lisp Object System).

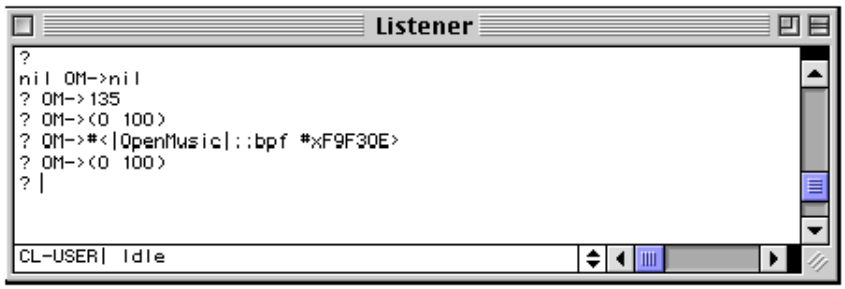
The Listener Window

OM runs on top of the Digitool MCL. OM Communicates with the user via Digitool MCL, and Digitool MCL communicates with the user via the Listener. Results from the evaluation of OM patches are printed here.

When the Listener is the active window, you are effectively in Digitool MCL, as evidenced by the change in menus at the top of the screen. Errors encountered while running a patch are printed in the Listener. For example:



When errors of this kind are encountered, patch execution stops and the Listener (i.e., MCL) waits for the user to choose a restart from the Lisp⟶Restarts menu. Sometimes an error will even occur in a second Listener window which appears. While MCL will continue to allow you to manipulate objects in OM while waiting for a restart, this is not advisable. Click in the Listener, choose Abort from the Lisp menu.The Listener should clear, and the last thing written in the listener should be a question mark, which indicates you can continue working with OpenMusic:


Listener
?
nil OM->nil
? OM->135
? OM->(0 100)
? OM->#<|OpenMusic|::bpf #xF9F30E>
? OM->(0 100)
?
CL-USER| Idle

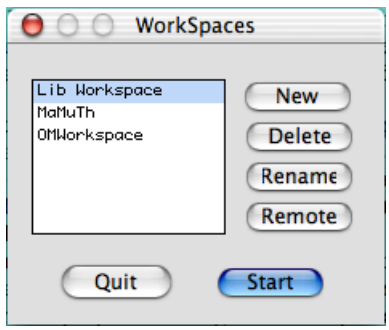
Chapter 2. The Workspace

The main interface to OpenMusic is the Workspace window. The Workspace is a icon-oriented browser which behaves in much the same way as the Macintosh Finder. Icons represent OpenMusic objects or folders where objects (or subfolders) may be stored.

Objects the Workspace are persistent. This means they hang around until you delete them, which is done with the Clear command. OM performs a general save of the Workspace and all objects in it at the end of each session. Nonetheless, you may force a save of any part of the Workspace by using the Save command. If you are in a patch, only that patch is saved. If you are in the Workspace, you may either select an object and save it, or click in the background so that no object is selected and then save, which saves the entire Workspace. This is useful since there is no undo function in OM. It will also save you a good deal of stress in the event of a crash.

Managing Workspaces

The Workspace is the top level of the OM environment. Everything you do happens in the workspace. Multiple workspaces can be maintained so that several users can use the same computer. When you start OM, if there is more than one Workspace present, a dialog box will appear asking which Workspace you'd like to work in. If you go to the OpenMusic folder, you'll notice a folder called `Workspaces`. Each subfolder within this one is a Workspace you can choose to work in at the startup dialog. While the folders in the `Workspaces` folder represent your Workspaces, they are manages by OM and the contents should not be modified directly, except in case of disaster.



From this dialog you can also choose to create a new Workspace, delete a Workspace, or rename one. The Remote option allows you to work in a Workspace which is not in the Workspaces folder (see below).

I recommend doing two things in this folder. Firstly, make a copy of the folder `OM Workspace` and call it "Empty" or "Blank" or something- just keep this copy around in case you want access to a pristine Workspace without reinstalling from the CD-ROM. Secondly, make a back-up copy of whatever Workspace you use. Command-**d** makes a duplicate of a folder and adds 'copy' to the end of the filename. At the end of each session, throw out your old copy and create a new one. This way if anything goes wrong during your session you can throw out your Workspace and make a new one from the back-up. This is important because unfortunately **there is no Undo function in OpenMusic**.

The Menu Bar

The Workspace has its own menu bar at the top of the screen, different from what you'll see when you're in a patch or in the Listener. The following are the most relevant selections:

The File menu

Under New, you may choose to create new folders, patches or maquettes. There are keyboard shortcuts, which are listed.

With Save you can force a save of the current patch, or of the entire Workspace, as described above.

Duplicate makes a copy of the item(s) currently selected. It is equivalent to option-dragging the item(s).

Make Alias is a special command used to create copies of class definitions for subclassing, during the creation of a new class definition. It's use is described in the section Classes.

Preferences brings up the Preferences pane, described in its own chapter.

The Edit menu

The usual Mac commands Cut, Copy, and Paste can be found here.

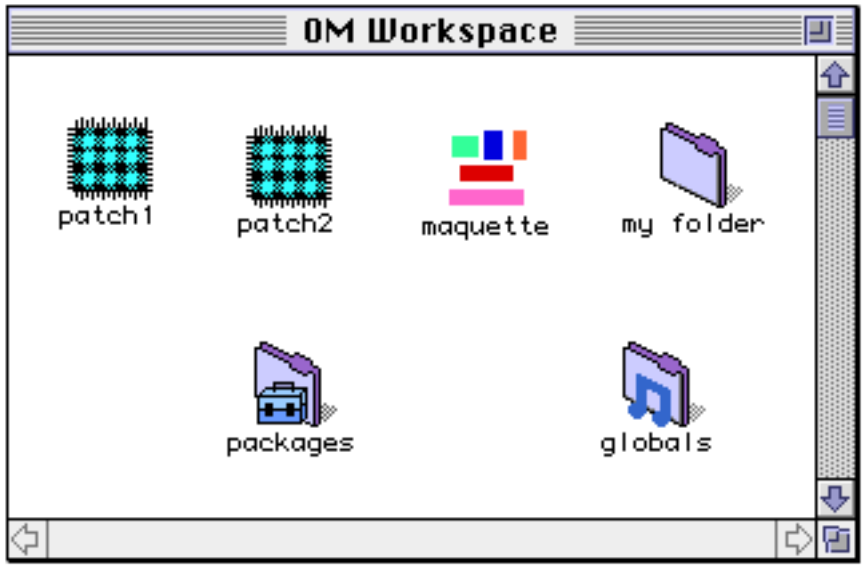
Clear is used to remove items from the Workspace. You must use the menu item, as the **delete** key will not work.

Presentation

This controls how the Workspace looks. The default setting is icons, but you can also have OM display your objects as a list, sorted by either name or type.

Objects in the Workspace

The Workspace contains patches, maquettes, and folders. Two of these folders are special folders: `globals` and `packages`. They cannot be deleted. The `packages` folder holds all of the function and class definitions for the environment, organized into a series of suitcases. See the chapter Packages for more info.



Manipulating Items in the Workspace

Drag & Drop

The Workspace supports drag & drop actions, just like the Mac finder. The standard mouse actions of **shift**-selecting multiple objects and dragging a rectangle to enclose objects work as well. You may copy objects by holding option as you drag them.

Import and Export

Drag & Drop can also be used to pass objects between the Finder itself and OM. Dragging an OM object to the finder creates an export copy of that object in the finder, which can then be dragged back into any OM Workspace. In addition, some file types (MIDI, for example) can be imported directly into OM by dragging from the finder into a patch window.

Contextual menus

Holding ctrl while clicking the mouse brings up a contextual menu. The items on this menu depending on the where you clicked. When you click on the background, you may choose from among the following:

- import file - same as drag & drop importing of a file from the finder, except that you locate the file in a standard Mac OS dialog box.
- import folder - same as above, for folders.
- view - duplicates the View menu in the menu bar.

When you click on an object, you may choose from among the following:

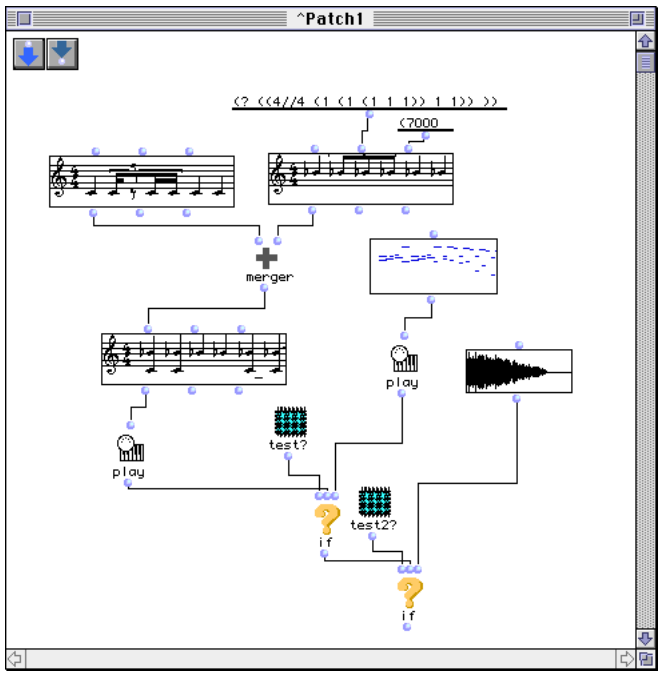
- open - opens the object.

- get info - brings up any information added about the object.

- duplicate - creates a copy of the object, same as option-dragging it.
- export - same as drag & drop export of an object to the finder, except that you choose the save location in a standard Mac OS dialog box.

Chapter 3. Patches

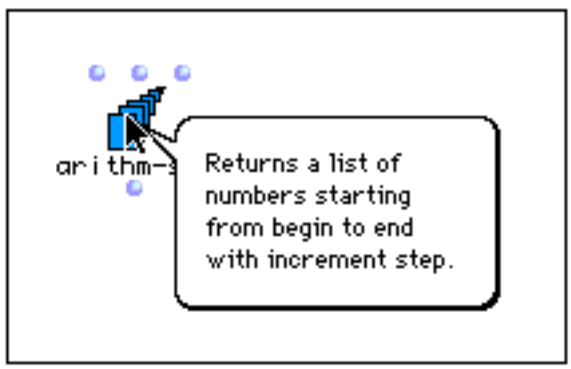
A patch is the basic unit of OM object-oriented programming. Within a patch, you create a flowchart which graphically describes *LISP* code accomplishing a specific function. The primary building blocks of patches are functions and classes although other objects are used, albeit more rarely:



Here we see some of the possible contents of a patch: OM musical objects with a graphic view of their contents, OM functions `merger`, `play`, etc., numerical values (7000 and the rhythm tree above), external MIDI and sound files with graphic representations of their contents, and other patches `test?` and `test2?`.

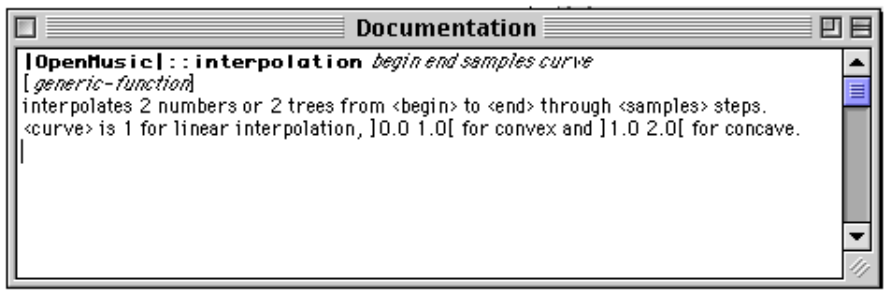
The in-line documentation

Within OpenMusic, four types of help are always available to the user. The first is a pop-up bubble help. This is accessed by command-clicking on any object or any input or output of an object. When you do this on an object, a description of the object pops up:



...and when you click on an input or output, its name is displayed, along with a decription of the input data types, if available (this information is not always present)

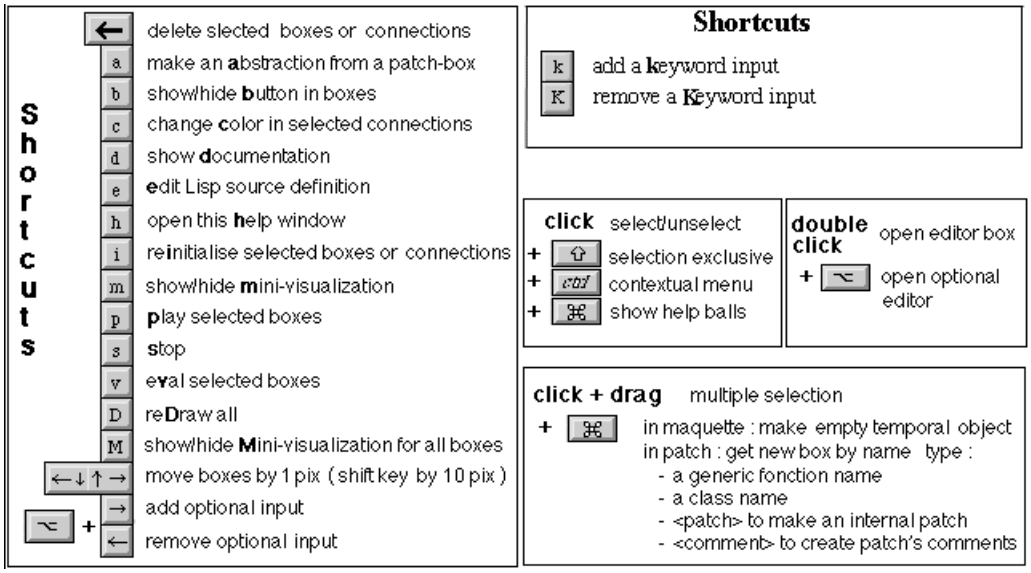
The second type of help is a more complete on-line doc, which is brought up by selecting an object and hitting **d**. A window with any available documentation pops up. For example:



Note: This built-in help can be a bit dicey. Eventually, it will be brought up to date in the same way as this manual. For now, it's a little like playing the lottery. You may get just the names of the inputs. You may get the information you need. You may get the information you need but in French or another language. Or you may get nothing at all. Always check this manual if you don't find what you want.

There is also an information window describing the type of object that can be brought up by selecing the object and typing command-**i**.

Finally, there is also a list of keyboard commands available in the patch window by hitting **h**. Note that these commands are case-sensitive. **M** and **m** are different commands. Check that capslock is off if you're having problems.



Patches in the Workspace

Patches in the Workspace are represented by a patch icon:

Within the Workspace, this icon represents the master copy of the patch; it may be moved around the Workspace but there can be only one patch icon for a particular patch. If you make a copy of the patch it represents a distinct patch with no connection to its parent. When you drag a patch into a patch window, you create a reference to this master code. You may therefore have as many references to the patch as you wish, in the form of patch icons, within the patch window. Double-clicking any of these brings up the patch editor window. When you change the patch in this window, you make changes to the master copy of the patch, and those changes are thus reflected in every reference (i.e. patch icon) in all of the other patch windows where it appears. If, therefore, you wish to make changes on the patch which will be local only, you have two options: create a copy of the patch in the Workspace and use that patch instead; or create an abtsraction of the patch (see below).

Patch structure

Patches may contain any number of other OM objects, including other patches.

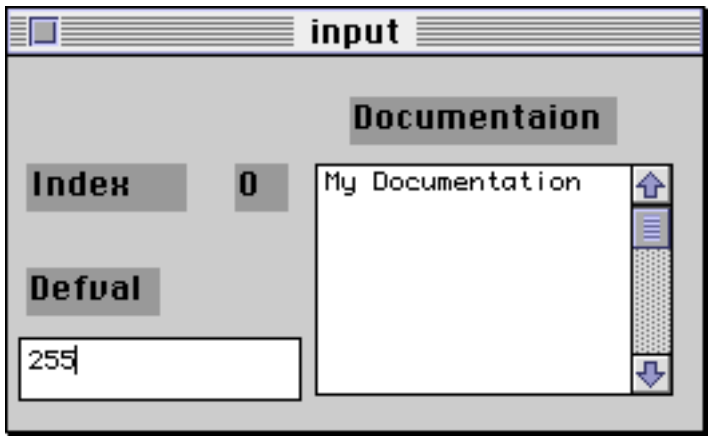
Don't leave errors unfixed!: Occasionally you will encounter errors while programming your patches. It is very important to fix the part of the patch that generated the error before quitting out of OM. This is important because when you open a patch OM reconstructs it from the save file. Reconstructing the patch can generate the same errors as evaluating the patch, and this can result in a patch which you simply can't open because it generates an error every time you try to open it.

Therefore, resolve the errors BEFORE leaving OM. If you can't resolve the error, disconnect or delete the objects producing the error. If you don't you risk losing the contents of the patch.

Inputs and outputs

The patch itself may have inputs and outputs allowing it to communicate with the outside world when used in other patchs. These are added with the two buttons in the upper-left corner of the patch window.

They are deleted by selecting them and hitting delete. They can be renamed by clicking on the name portion of the icon. Double-clicking an input icon brings up a window where you can edit the default value given at that input. Note that objects may be dragged directly into the default value field to set it.



When you use the patch in another patch, this default value will be passed by that input whenever there is nothing connected to that particular input. Additionally, this value is used if you evaluate *within* the patch, regardless of what is connected to it on the outside.

Additional inputs and outputs and changes to existing ones may not be readily visible on the patch icon from the outside. In this case you can use the contextual menu selection 'update doc' on the patch to update the icon. (See contextual menus, below.)

Objects within the patch

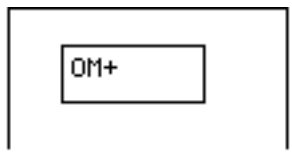
There are three methods for adding objects to a patch. In the case of functions and class factories, you may select them from the pull down menus Functions and Classes which appear when you are in the patch window. The cursor changes form and the object is placed where you click.

You may also drag objects into the patch window, either from the Workspace or the Finder itself. Any patch may be dragged into your patch window to create a reference to it. Any function or class definition from the `packages` folder may be dragged to the patch window to create a function reference or a class factory. option-dragging items within the patch or from

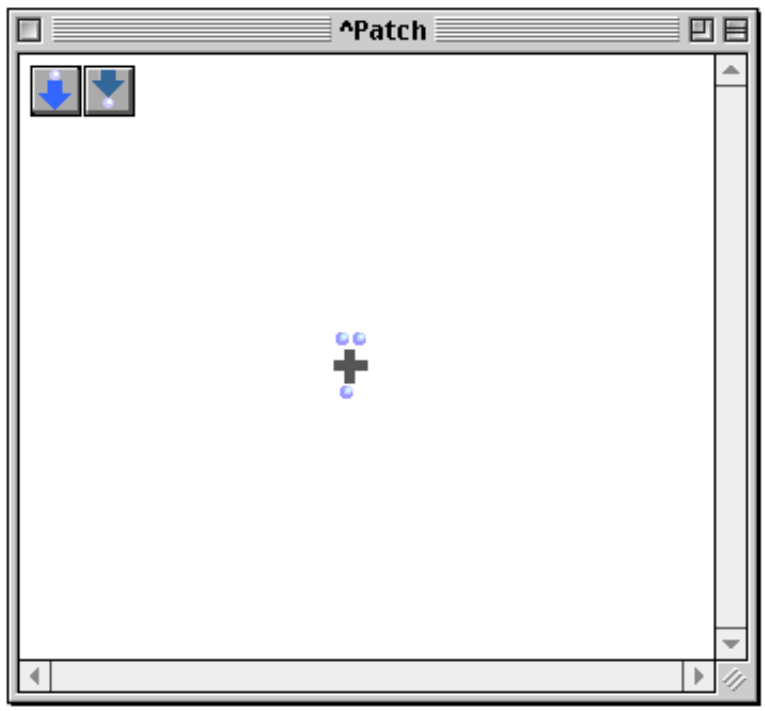
other patches copies those objects to the patch. You may also drag certain kind of files from the Finder itself. These include MIDI files, AIFF sound files, and OM objects saved to the finder (see the Workspace chapter).

The third method is the most convenient once you are familiar with the available objects and classes. Command-clicking on the patch window background opens a field:

...where you can enter text or numbers or a list. Press **enter**, not **return** after entering the information, or click elsewhere in the patch window. If you enter numbers or a list, OM assumes that these are data that you'd like to plug into the input of a function and creates a box to hold them. If you type text, OM attempts to match that text to the names of all the objects and classes in OM. If it finds match, that object replaces the field. Typing:



gets you



These data fields can be edited at any time by double-clicking them. Remember to confirm your changes by hitting **enter**, not **return**.

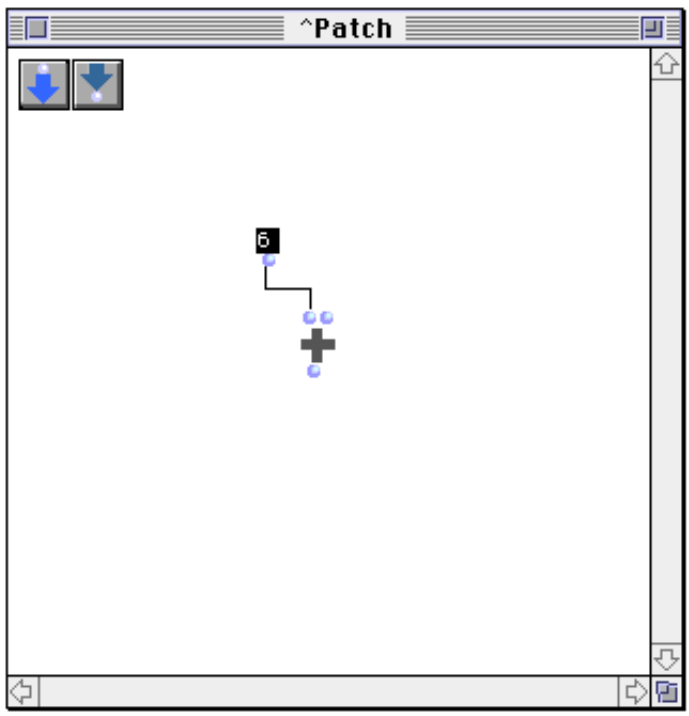
Making connections within the patch

To connect one object to another, simply drag a connection from any output to any input. While you are dragging, information bubbles automatically appear above any input the cursor touches in order to make choosing an input easier. When you release the mouse button, a 'patch cable' connection is drawn. To remove a connection you may either click on the patch cable line and hit delete, or option-click the input to which it is connected. Note that it is not necessary to delete connections before making a different connection to an input, as only one thing may be connected to a given input at a time.

The patch cable itself may be graphically adjusted by clicking on it and dragging the square handles which appear. This has no effect on the patch, although it will prevent automatic adjustment of the patch cable if you move either of the objects to which it is connected. While it is selected you may also change its color by hitting **c**, which cycles through a pallete of available colors. Shift-clicking to include other patch cables allows you to do this to all of them at once. Color-coding your patch cables inproves visibility and can help you understand what's happening in a complicated patch you haven't looked at in a long time.

When you move the mouse pointer over an input which is not connected to anything, the default value for that input pops up. This value is encoded with the function or class, but may be changed for the present icon only by clicking the input and entering a new value in the field that appears.

In addition, shift-clicking an input puts the default value in a newly created data box and creates a connection to that input automatically, a real time-saver:



Evaluating patches

To get the results of a patch, you must evaluate it. This is done either by option-clicking the output of the function or class you'd like to evaluate, or by selecting an object and hitting

v, in which case it is automtatically the leftmost output that is evaluated. The results are returned in the Listener, and any graphic views in the patch window are updated. Evaluating a patch is usually done at the last item in the chain, but you may see intermediate results by evaluating any link in the chain.

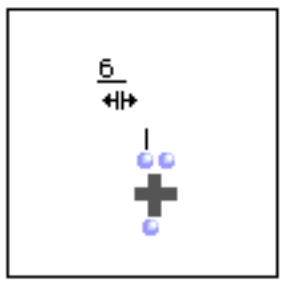
Variables

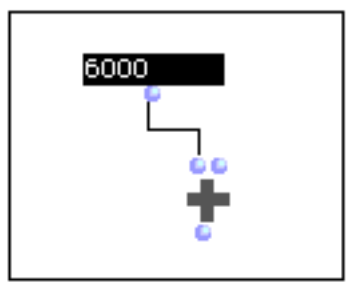
Certain kinds of outputs can be used to create 'variables'. A variable is like a snapshot of the output it was taken from. This variable will always return the value it was created with when evaluated. It's a useful way to save interesting results of an operation. Variables may be not be created at outputs yielding LISP "primitives" (i.e. the classes in the package `kernel:lisp` kernel) with the exception of lists. Otherwise, any output may be used to create a variable. Variables are created by holding shift while option-clicking the desired output. See the section on Classes for more information.

Manipulating icons in the Patch Window

Icons can be dragged with the mouse to move them around. They can also be moved with the cursor arrow keys. Holding shift while using the cursor keys increases the distance of the 'nudge.'

The size of any icon can be changed by moving the cursor near the bottom-right part of its box. The cursor will change to a small arrow. You may then drag the icon out to any size you wish, useful for de-crowding inputs and outputs.





Abstractions

As mentioned earlier, the patch icon, when used in a another patch window, represents a reference to the master copy of the class kept in the Workspace, and any changes made to the patch are made to the original master. If you don't want to do this, you must either make a copy of the patch in the Workspace and use that instead, or you may use an abstraction. Abstractions may be made of patches and **Maquette**s. Abstracting creates an exact copy of the object that exists only locally, i.e. it has no master copy in the workspace. Any changes you make to this object only affect the object itself. Abstractions are represented by red versions of the icons they were abstracted from:

Abstracting cannot be undone. You can, however, create a new patch and drag the entire contents of the abstraction from its window to the patch window. Inputs and outputs must be recreated from scratch in the new patch window, since they cannot be copied.

Abstractions may be created from scratch by command-clicking in the patch window background and typing **`patch`** for a patch abstraction and **`maquette`** for a maquette. Patch abstractions are useful for organizing complicated patches. Putting related sub-functions in a specially created and named subpatch abstraction makes your patches easier to understand and reduces clutter.

Contextual menus

Holding ctrl while clicking in the patch window or on an object brings up a contextual menu. On the patch window background, this menu has two choices: comment allows you to add a comment box, which contains text of any kind about the patch. The other option is last saved, which reverts the patch to its last saved state. Useful because there is no Undo in OM.

On patch objects, the option update doc checks for new or deleted inputs and outputs, changes to input definitions and documentation within the patch, and updates the patch icon. This is sometimes necessary when adding new inputs to a patch you are using in another patch.

On other objects, you have the option to evaluate the patch, to show or hide the object state button (see below), to copy the item (equvalent to option-dragging it), to get info on the object (equivalent to selecting it and hitting command-**i**), and import or export an object from the Finder (also possible with drag & drop - see the chapter on the Workspace for more information.

Object states

Objects (both factories and classes) in OM have five modes, or states, which affect the results of evaluation. In its normal state, a box returns the results of whatever operation it performs (for functions) or slot you evaluate (for classes). In addition, there are four special states. They are activated by selecting an object and hitting **b**, or by choosing s/h boxes from the contextual menu. When you do this, a small box appears in the upper left corner of the icon representing its special state. You can cycle through the four states by clicking the small box, or return the object to its normal state by hitting **b** again. The four states are:

- **Locked** - The object is locked and will not recalculate its outout. In the case of functions, it will return the last value calculated when evaluated. In the case of factories, it will return the last instance produced through evaluation or editing of the contents in the graphic editor. This state is primarily used to preserve the contents of edited musical objects.

- **'Eval-once'** - The object only calculates a result for a given evaluation, even if the object's output is connected to multiple inputs of other objects or in a loop. This state is primarily used with objects with random elements which pass data to multiple inputs so that each input receives the same value and not a re-evaluated (and most likely, different) value from the random function.

- **Lambda** - The value returned is not the result of the calculation but rather the object itself as a function. This state is primarily used with functions like `mapcar` and `funcall` which apply another function to data. Note that when used in a calculation by another function, the empty inputs in a lambda mode function always pass their default values as set in the function definition and if you want to change these you must connect data boxes to the inputs of the lambda function, not simply modify the defaults at the input concerned.

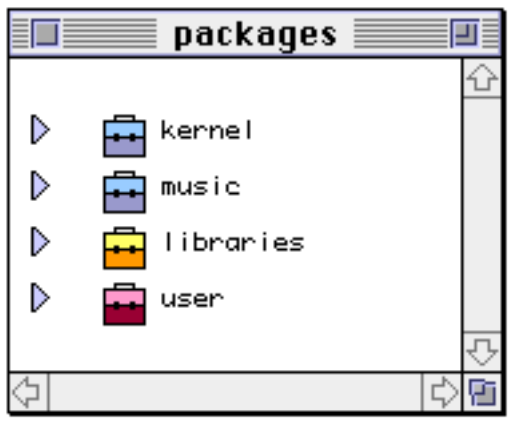
- **'Itself' or Reference** - the value returned is the OM reference pointed to by the icon, i.e. the digital address of the object in the OM code. If used with a patch icon, the patch object itself is passed. If used with a function icon, the CLOS generic function object is passed. For a factory, the class definition it references is passed (if you don't understand these, don't worry about it.) This state is primarily used to work with **Maquette**s, in order to add or remove or submaquettes from them.

Chapter 4. Packages

The packages folder is the place where OM function and class definitions and classes are stored. Classes are the models from which new objects are created. Functions are operations performed on classes. Both classes and functions are represented in by icons, which can be dragged from the packages folder to the Workspace for use.

The package concept is inspired by the Java *programming language*. It is a place where related functionalities are stored together. A package is represented by a suitcase icon with two parts, an upper, lighter colored part, and a lower, darker colored part. These two parts give access to different aspects of the package. Packages can have subpackages. You'll be able to create your own subpackages in the user package, and fill them with your own classes and functions.

Though they may contain any number of packages, all OM packages folders must contain at least the four basic packages: kernel, music, libraries, and user. Double clicking opens the packages folder:



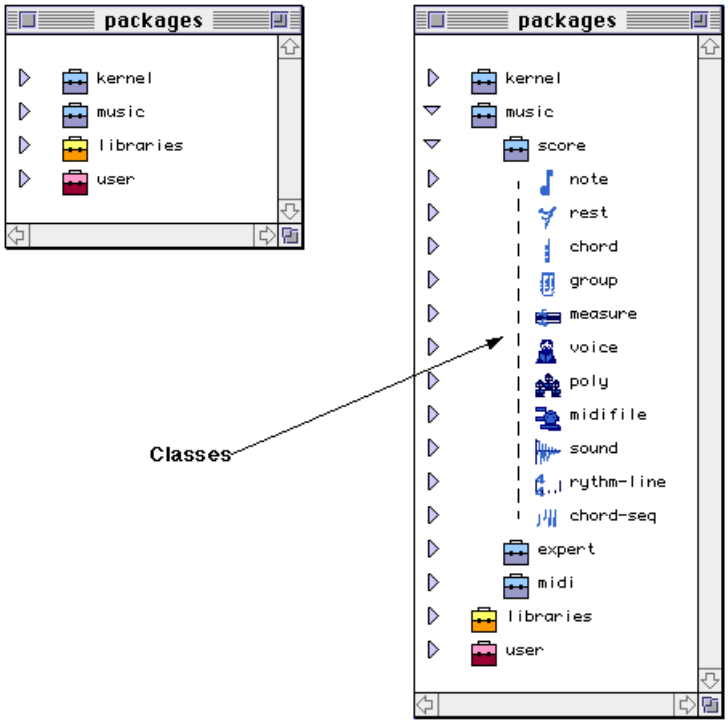
Kernel stores functions and class definitions for the LISP kernel, the foundation on which OM is built. Here you will find the data types and functions for mathematical calculations, basic data processing and list management.

The music suitcase stores the functions and classes particular to OM, i.e. all the playable musical objects and the functions which act on them.

The libraries folder holds classes and functions for any libraries available to OM. A library is a stand-alone module of class and function definitions grouped into a suitcase. Before you can access these functions and classes, you must load the library by double-clicking the lower half of the suitcase icon, after which the suitcase opens and you can view the contents. The functions and classes of the suitcases will also become available in the Functions and Classes menu bar when in the patch window.

The user folder holds class and function definitions created by you, O great user.

In order to view their contents, packages may be expanded with the icon to their left or opened in in a separate window by clicking in the **lower**, darker portion of the suitcase icon.

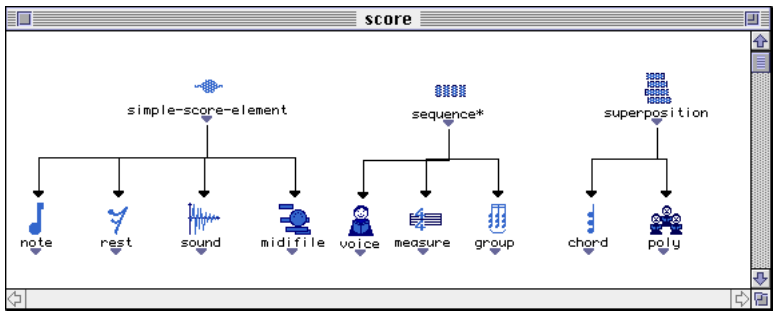


Managing Classes

When you open a suitcase you will see any class definitions it contains. Some packages contain only function definitions, an so you'll see no class definitions. If there are class definitions, they will be represented by icons. Adding subpackages is done here with the File⟶Add Package command. Deleting classes is also done here by selecting the icon representing the class definition and choosing Edit⟶Clear.

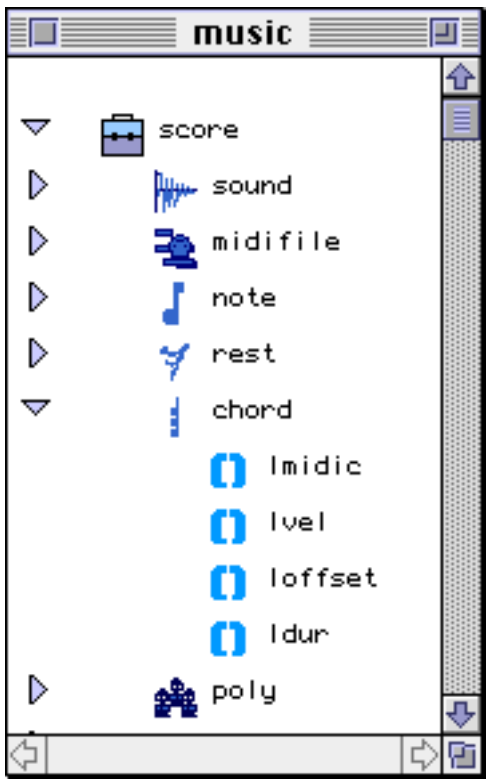
Inheritance

Opening a suitcase is done by double-clicking the lower half of the icon. Double-clicking the upper half, however, brings up a window with a graphic representation of the inhertance heirarchy for any class definitions in the suitcase. If the suitcase contains only subsuitcases or the class definitions it contains do not inherit from other classes, this window will be blank. New classes are created here with the File⟶New Class command. Subclassing (for the creation of user classes which inherit from OM objects is done in this window by creating an alias of the icon of the class from which your new class will inherit (with the File⟶Make Alias command of the menu) and then dragging it to the class inheritance window of the user package and connecting it to the icon representing the class you are creating. See Classes for more details.



Slots

To view the slots for a class definition, click the arrow icon to its left to expand it. Some classes (integer or list, for example) have no slots, and so there will be no arrow icon. When expanded, each of the class slots will be listed along with an icon representing the data type accepted at that slot.



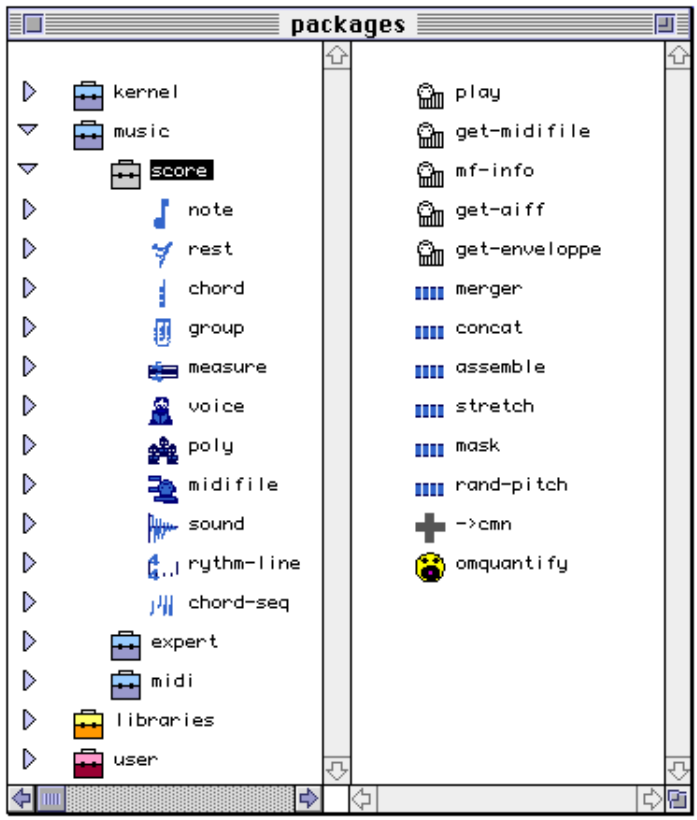
You may edit, add, and delete slot definitions in the slot list window for user-defined classes (the OM classes are locked and cannot be modified). This window is accessed by double-clicking the icon representing the class definition, either in the package window or the inheritance heirarchy window. A list of all the slots you've added to the class appears (slots inherited from other classes do not appear). You may choose whether the slot will apply to instances of the class or whether it will exist only within the class definition (for subclassing, for example) by choosing a method from the Allocation menu. If you have chosen instance, you have the option to include the slot graphically in the factory icon for the class by checking the Show. You may also edit the default value for the slot by double-clicking the value shown in the Default value column.

Initialization methods

You can edit the initialization methods for your user-defined classes by option-clicking the class icon in the package window. A new column with an icon representing the initialization method appears on the right. Double-click this icon to open the initialization method editor. See Classes for more details.

Managing Functions

In order to view the function definitions stored in a particular package, you may option-click on either half of the small package icon. A new column opens on the right, displaying icons representing all the function definitions of present in the package:



Method definitions

You may option-click on a function icon in the package window in order to see a third column with the method definitions displayed, represented as the the function icon itself combined with a series of other icons representing the data types processed by that method.

In order to add or edit methods, you must double-click the function definition icon, bringing up a new window with the method definitions. They can be opened and edited by double-clicking them. Adding new methods is also possible with the File⟶New Method. To clear a method, select it in this window and choose Edit⟶Clear.

Chapter 5. Globals

The `globals`, along with `packages`, is one of two special folders in the Workspace. It cannot be deleted.

The `globals` folder stores variables (i.e. class instances) that you want to share among patches. This is an advanced feature that is not covered in this manual.

Chapter 6. Data Types

Data type is an important concept in OM. OM function and class inputs will only accept certain data-types. For example, attempting to give a **Note** object a list of midics at the *midic* input will generate an error. This is because the *midic* input is *typed* to accept only numbers, and list is not a type of number. Some inputs can accept more than one data type, in which they case they select one of a set of internal tools called *methods* to deal with that particular data type.

The idea of data type is related to and overlaps somewhat with the concept of Classes. A class defines a data structure, which is itself an organization of data types used as a model for producing objects. The class **Note**, for example, contains information about the note's pitch, volume, and midi channel. The ensemble of these elements is the class definition. When a class is used as input for a function or another class, the class is a data type. Indeed, some OM functions will only accept one kind of OM music object class as their data type.

For more on data types, see *the glossary entry* and the table in the Introduction to the Function Reference.

Chapter 7. Rhythm Trees

Rhythm trees are a special type of *nested* list used to represent rhythmic structures, primarily for use in the **Voice** object. They are an alternative to traditional music notation. The rhythm tree is inspired by the structure of *LISP* itself, a series of nested lists. The advantage of a rhythm tree is that it represents the totality of the hierarchical relationship of traditional rhythmic notation (measures divided in to groups divided into beats divided into notes) in a numeric form, unlike the simple lists of durations and onsets of the **Chord-seq** object, for instance. The disadvantage of the rhythmic tree is that the symbolic form, while very convenient for computers, is somewhat less convenient for human beings. They can contain ridiculous numbers of parentheses that will test the eyesight of even the most assiduous user. Making sense of rhythm trees requires a complete understanding of their structure. Users are therefore advised to complete the tutorial on rhythm trees and then come back to this section for the most complete picture possible.

The structure

A Rhythm tree is a series of nested lists which all have the same form: **(D S)**, wherein

- *D* is a quantity expressing a duration (relative, not absolute).
- **S** is a list of items defining a series of subdivisions of the duration *D*, each item of **S** being either:
  - a number
  - a sublist, itself having the same structure **(D S)**.

The actual duration indicated by *D* depends on the group of which its list is a part. At the highest level, the basic unit is assumed to be the whole note. So,

```
(1 (1 1 1 1))
```

is a structure whose duration is a whole-note (1) which is subdivided into 4 equal parts, which would be 4 quarter notes in this case. Similarly,

```
(2 ( (1 (1 1 1 1)) (1 (1 1 1 1)) )
```

is a structure with a duration of two whole-notes, containing two equal substructures, each lasting a whole-note. These substructures in turn contain 4 equal values i.e. quarter notes. So it can be interpreted as a voice containing 2 measures in 4/4.

As mentioned, the *D* element is measured in whole-note units. The first element of a **Voice** object, for example, will therefore be the duration of the sequence in whole notes.

Sublists on the first level of the tree represent measures. When the quantity *D* is at the level of a measure, it is still expressed in whole note units, but by convention it is expressed in a more intuitive form closer to that of a time signature. This may either be a fraction i.e.

```
4/4 7/8 5/2
```

, or a list of two elements, i.e.

```
(4 4) (7 8) (5 2)
```

.

Note that fractions in LISP are always automatically reduced to their lowed common denominator by the machine, which turns a time signature of 4/4 into 1/1. To avoid this you can use a special double-slash notation:

```
4//4 7//8 5//2
```

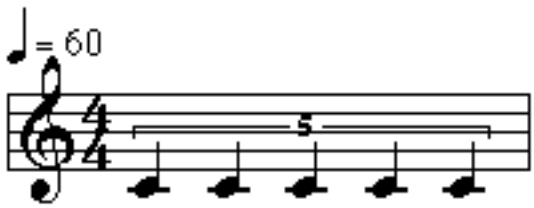
. Otherwise use lists.

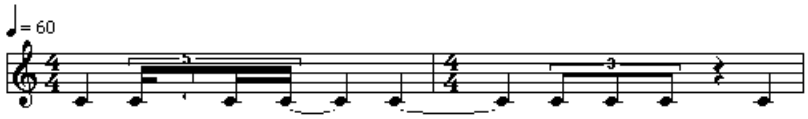
At the very top level of the tree, it can be difficult and tedious to calculate the duration element *D* ot the entire sequence. On the highest level and on the highest level only, *D* can be replaced by a question-mark '?'. In this, case OM will automatically figure out the actual value of *D*. Our previous example could thus have been written as:

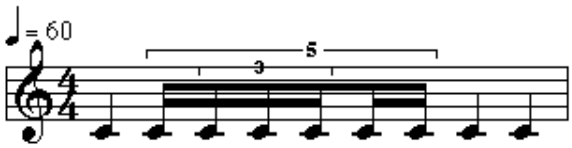
```
(? ( (1 (1 1 1 1)) (1 (1 1 1 1))) )
```

Some Examples

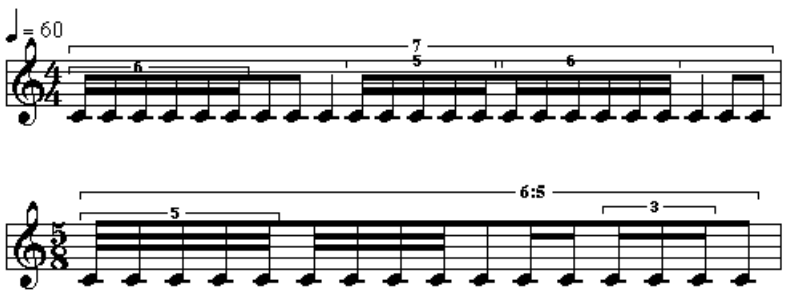
```
(1 (1 1 1 1 1))
```

considered as a single group. The choice of the quarter note here to represent the first 1 is arbitrary; this list could have taken place at any level of the tree:

```
(? ((4//4 (1 (1 (1 -2 1 1)) 1.0 1)) (4//4 (1.0 (1 (1 1 1)) -1 1))))
```

considered as a voice. Note the question mark telling OM to figure out the actual voice duration. Notive also that there are some negative numbers and some numbers with a decimal point. The negative numbers are rendered as rests, and the floats (numbers with decimals) are rendered as tied to the previous note. Note also how the quintuplet is notated. The list (1 -2 1 1) represents five elements evenly dividing a quarter note. (Because the sum of the digits is 5, not counting minus signs) The second and third values are combined in a single event, signified by the use of the sum of 1 and 1, 2. The two is negative, meaning that it is a rest.

Here we put a triplet inside a quintuplet.

```
(? (((4 4) (1 (1 (1 (2 (1 1 1)) 1 1)) 1 1))))
```

And another example using nested irrational pulses. Note how we notate the time signatures as their own little lists.

```
(? (((4 4) ((4 ((1 (1 1 1 1 1 1)) (1 (1 1)) 1 (1 (1 1 1 1 1)) (1 (1 1 1 1 1 1)) 1 (1 (1
```



Chapter 8. Functions

Functions in OM can respond to more than one data type. Take om+, for example. When you give it a number and a list, it adds the number to all the elements of the list. However, you can also give it two numbers, in which case it simply adds them. How does it know what to do?

Functions and methods

It turns out that within the function om+ are a series of sub-functions called *methods*. A method is a set of instructions for processig a specific combination of data types in a function. When you use om+ with a list and a number, om+ automatically calls a method for adding a list and a number. You can see the methods of a function by opening its box with a double-click. If the function cannot find a method appropriate to the data types you are trying to give it, it generates an error.

A function composed of methods for different data types is called a generic function. **All OM functions are generic functions.** The advantage of working with generic functions is that operations which are conceptually similar but apply to widely differing data types can be organized into a single function with a single name, which is much more logical for humans. When you define functions in OM, then, you are really just defining a name.What you implement with the graphic interface are the methods.

Functions in OM are also multivalued. This means that they can yield more than one result. This is symbolized by multiple outputs on certain function and class icons.

Function definitions

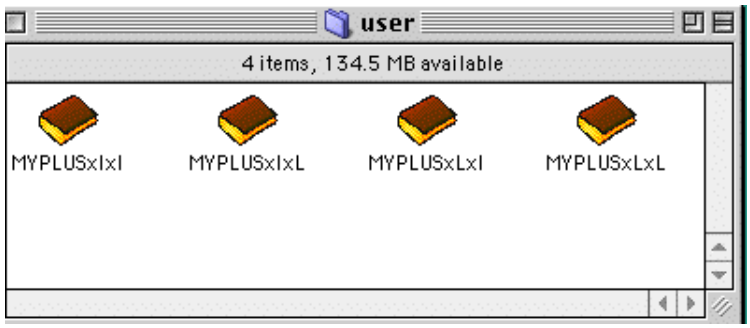
All function definitions, including functions defined by you, are stored in the packages folder. Functions are managed here, as detailed in the chapter on the packages folder.. You may delete a function definition (and all associated methods) by selecting the function definition icon and choosing Edit⟶Clear. Functions are created in the Patch window itself by choosing File⟶New Generic Function.

Dead methods

As mentioned earlier, a function icon is a reference to a master copy of the function in the packages folder. If the master copy is not present or not found, the function icon is replaced by the special dead method icon in all the patches in which it is used:

The dead method cannot be evaluated; nor can anything connected to it. The dead method indicates one of two things; Firstly, it's possible that the function definition it refers to was

deleted from the `packages` folder. Second, it's possible that the patch where the dead method is found was imported from a workspace without also copying the function to which the dead method refers. User function definitions can be found in folder for the Workspace you are using (see the chapter on Workspaces for information), in the subfolder `user`:



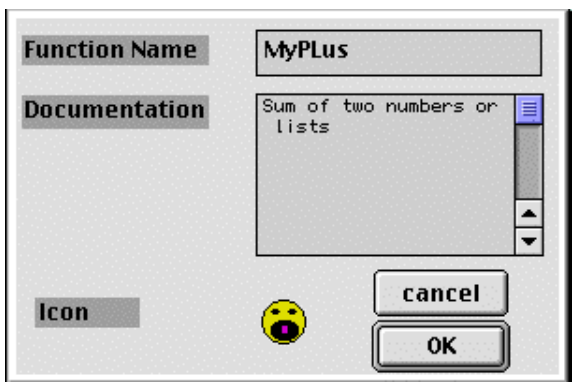
Each method for the function is stored separately and indicated by a suffix attached to the name of the function. Copy these files to the `user` directory of the Workspace you use in order to access the functions.

Creating new functions

When you create a function, you define a unique name and a first method for the function, which also permanently sets the number (not the type) of inputs. Afterwards, new methods may be added as needed. Follow these steps to create a new generic function:

Creating the function name and appearance

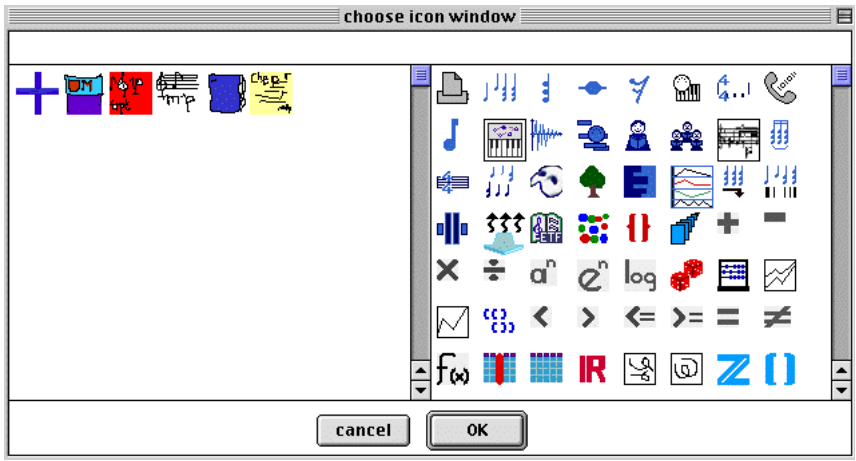
In the patch window, choose File⟶New Generic Function. The following dialog box appears:



Choose a name and write a description of the function (for the on-line help- see the Chapter on patches for more information.).

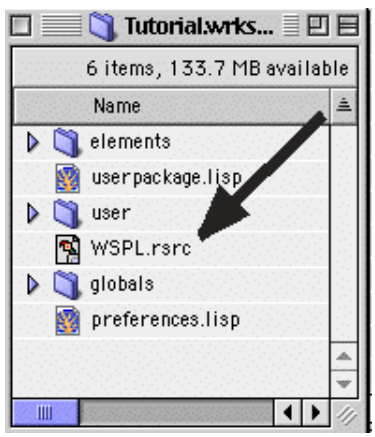
Important: Though OM will permit it, you shouldn't spaces in you function names, as it causes problems when trying to add the function to patches by the typing-in method.

Now choose an icon by clicking the icon itself and selecting one from the window that appears:



Icons on the right are standard OM icons. Icons on the left are user-customizable icons.

You may customize the icons using a resource editor such as **ResEdit** and adding these icons to the file `WSPL.rsrc` file in the current workspace folder.

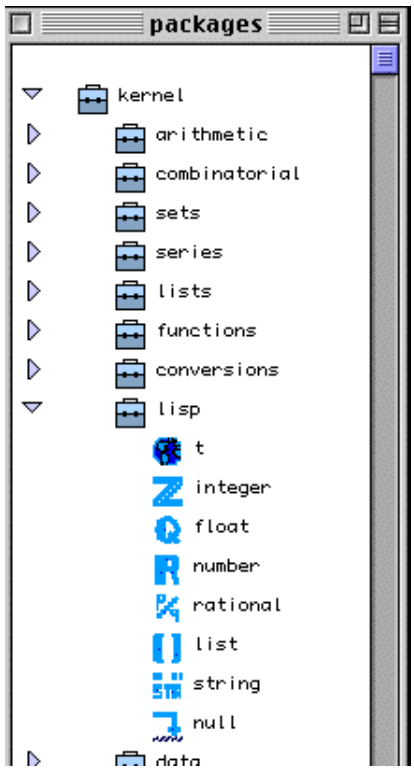


Be careful editing these resources. Always work with a copy!

Defining the first method

After you click OK, you wind up in a method definition window. This looks and acts a lot like a normal patch window. We can add inputs and outputs and combine other functions and patches in any way we like. The difference is that we must 'type' our inputs. When you add inputs to a patch, they are a universal type (in fact, they are the LISP type t) which can apply to any data type:

A method for type t will always be called if a more appropriate one cannot be found. You could leave it like this, but that would defeat the purpose of creating methods for different data types.In order to type our inputs, we drag an icon representing the data type we want to handle onto the input. These are all found in the `Packages` folder. We find all our functions and classes in this folder. In fact, any class is a data type when used as a function input. Here are shown the LISP package classes, but classes from any package can be used to type inputs. Rememeber that though you will be able to add methods at any later point in time, the number of inputs and outputs is fixed at function definition time and cannot be modified.



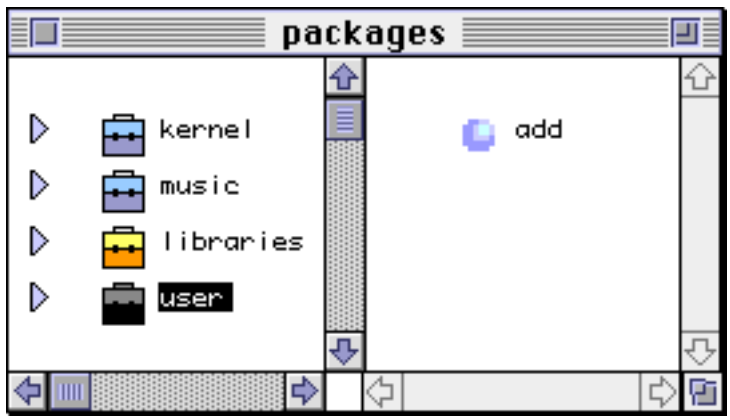
Drag the icon of the data-types (class definitions) you wish to use to the input icons in your patch window. The icons change to represent the new typing.

At this point you may also double-click the inputs to set the default values and write a documentation entry for the inputs, as you would do in a patch window. See the chapter on Patches for more information.

Now you can design the method exactly as you would any patch. All OM objects are fair game.

Adding additional methods

Once you define the function, it will appear in the packages folder. Option-click the user suitcase to reveal the functions within:



Double-clicking the function definition icon brings up a window where its methods are listed. Each method has the icon for the function itself, with a series of other icons added in order from left to right, representing the data types it handles at its inputs. Deleting methods can be done here by selecting them and choosing Edit⟶Clear.

To add a new method, choose File⟶New Method. Another method definition window comes up. The radio buttons across the top represent advanced functions that are not discussed in this manual. Leave primary checked. Notice that inputs and outputs cannot be added or subtracted as they were defined when you first defined the function. Type your inputs as before by dragging the appropriate class definition icons from the packages folder, and design the new method as you would any other patch.

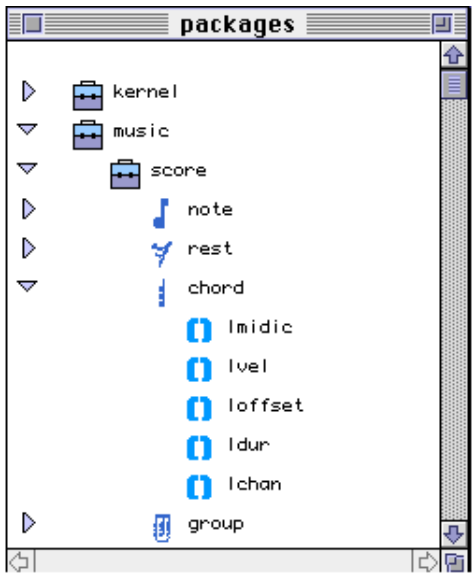
Close the window. The new method is defined and appears in the method definitions window of the packages folder.

Chapter 9. Classes, Instances, and Factories

Class is an important concept in *object-oriented programming languages*. A class is a a prototype for a certain kind of object. It is a definition of a data structure. It groups together any number of data items in a single object. Classes in OM include whole numbers (integers), decimal numbers (floats), lists, **Chord**s, **Voice**s, and many others.

Class Definitions

OM Class definitions are kept in the `Packages` folder in the Workspace. You can expand them by clicking on the down arror next to the name. You'll notice that Classes are grouped according to the part of OM which defines them. LISP functions are in the Kernel, and most of OM's music objects are in the Music:Score package:



You'll can view the internal structure of the class by expanding it in this view. As with the **Chord**, above, each class is composed of a number of pieces of data. Each place where data is stored is called a slot.

Slots

Slots are the internal components of classes. The **Chord** above has been expanded to reveal its slots: *`lmidic`*, *`lvel`*, *`loffset`*, *`ldur`*, *`lchan`*. Each slot holds a piece of data of the data type indicated by the icon to the left of the name. Notice that these data type icons are simply the icons from the classes of the **kernel:lisp** package.

Factories and Instances

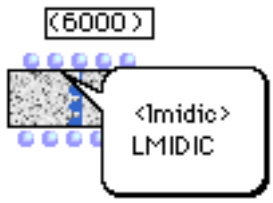
When you place a class icon in a patch window, either by dragging it from the Workspace or entering its name directly, you create what is called a *factory* box. The class is a definition

of a data structure, not a data structure itself. When you evaluate a factory, you create a data structure following the class definition. This data structure is called an *instance* of the class. A factory produces instances of the class it is associated with when you evaluate it.

A mini-visualization mode allows you to see 'into' the factory icon from the patch window. It is activated by selecting the factory icon and hitting **m**.

Ins and Outs

A factory box in the patch window has an equal number of inputs and outputs. The first (leftmost) input and output are always called *self*. *self* represents the entire object with all of its slots. Evaluating self yields a class instance. The other inputs and outputs correspond to the slots of the class, as defined in the class definition. Here, a **Chord** icon has been dragged to the patch window, creating a **Chord** factory:



The pop-up bubble is for the second input, *lmidic*. When you evaluate the class, the inputs of the class are evaluated and a new instance of the class is created. If connections to the inputs are present, this data is used in the instance, provided it is of the right type. If not, the default value for that input is used.

The default value of a slot is shown when you put the mouse pointer over its input on the factory icon. You may change the value of this default input by clicking the input and entering a new value. This default will then be used (only at this factory) for all future instances. Connecting a value to this input overrides the default value.

Most classes have a *self* output and input which allow you to pass or set all of their values as a group by making a single connection; this is easier than connecting all of the inputs of the object to their corresponding outputs on another object. If you have things connected to both *self* and the other slots of a factory, *self* is preferred. The instance will be created using the slot values of the object coming through *self* and the other connections will be ignored. Discconnect *self* if you need to use the other inputs. If you are trying to change

The *self* input accepts an entire object. This object may be of the same type as the factory, or it may be a class from which the current class can inherit data.

Inheritance

Inheritance is a characteristic of classes in object-oriented programming languages. Take the example of two imaginary classes, **publication** and **book**. A publication would have slots like *number of pages* and *editor*. All books have a number of pages and apublished. But a book might also have a single *author*. Rather than redefining the number of pages and the the editor of the **book** anew, the class **book** could *inherit* from the class **publication**. **book** would include all of the slots of **publication** with an extra slot, *author*. Other kinds

of publications (**pamphlet**, **magazine**) would also inherit from **publication** but have their own additional slots with information specific to them. Organizing classes into heirarchies of inheritance makes prorgams more logical and easier to write.

Inheritance can be put to work for you! OM music factory icons have the capacity to accept data directly from other classes via their *self* inputs. As an example, a **Note**'s *self* output can be plugged directly into the *self* input of **Chord**. A **Chord** is generated containing that note, because **Chord** knows what to do with **Note**s. Similarly, a list of **Note** objects is also acceptable at the *self* input of **Chord**.

You can find out what kinds of classes a given factory can accept at *self* from by using the online help in the patch window (accessed by selecting the object and hitting **d**.)

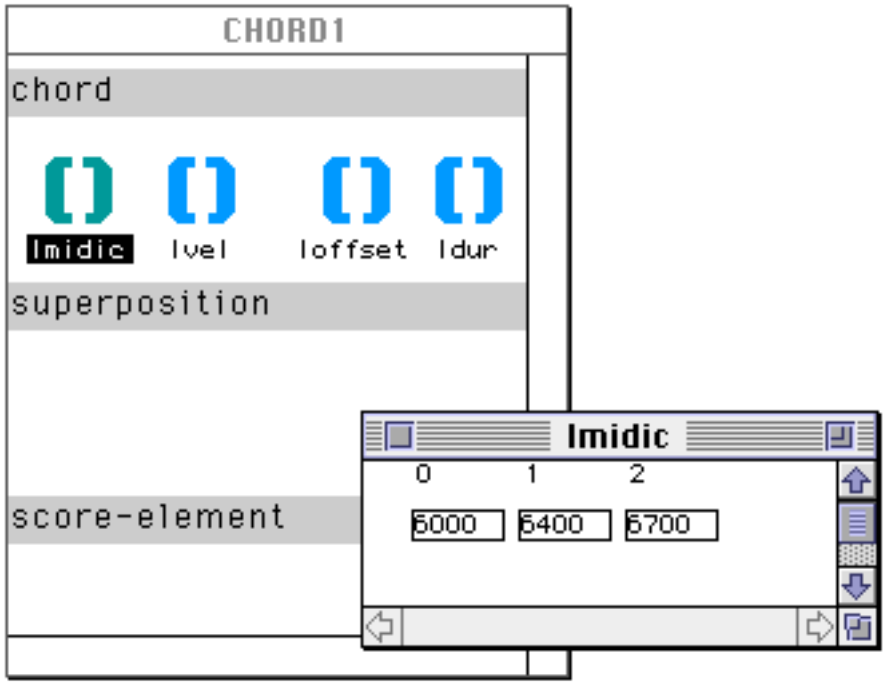
> **Note:** Though passing a **Note** to a **Chord** is sort of like an example of inheritance, this is not really the case. **Note** *does* inherit, but from a class called **simple-score-element** (see below). When you pass *self* between factory objects, the factory is actually calling an internal method which translates the object.. **Chord** has methods for dealing with **Chord** (obviously) but also for dealing with **Note** and a list of **Note**s at its *self* input.

Variables

The output of a factory icon at the *self* output (or any other output producing a collection of values, i.e. list or other OM musical class) can be captured by shift-option-clicking it. This creates an instance that is separate from the factory:

This can be thought of as a fixed variable representing a particular instance of the class. The single output of this instance is equivalent to the *self* output of the class. Variables are useful for preserving interesting musical results of evaluations. Be sure to lock the factory before creating the variable if you have random elements, as this does cause evaluation.

You may edit the variable as you do the factory icon, by double-clicking too open the graphic editor associated with that class. Not all classes have graphic editors associated with them, however. There is a second kind of generalized editor which is accessed by option-double-clicking the variable or class. It looks like:



...and while it is primitive, it works for all objects and allows access to all slots. It also shows all layers of inheritance. The **Chord** class, for example, inherits from a more general **superposition** class, which itself inherits from the very general **score-element** class). You can edit the slots by typing in values directly or dragging objects from a patch.

Creating a new class

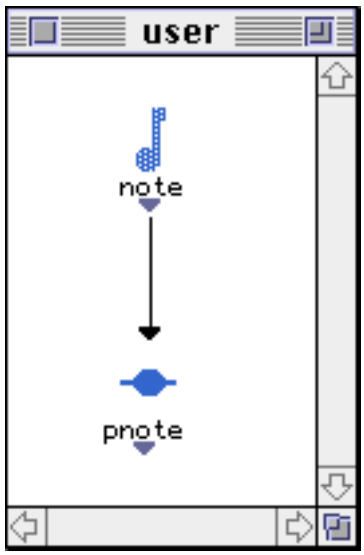
Here we create a user class named **Pnote**, a note that triggers a program change every time it is played. We'll tell OM that **Pnote** is just a **Note** plus some added information, by defining **Pnote** as a 'subclass' of Note. **Pnote** will inherit all of **Note**'s slots, and we'll add an extra slot to hold the program change.

Setting the inheritance

Here's how to do it:

- Open the **User** package class hierarchy window by double-clicking the upper (light) part of the user package icon.
- Choose File⟶New Class
- Enter a class name (**Pnote**), a documentation string, an icon in the dialog window displayed.
- Open the Music:Score package class hierarchy window by double-clicking the upper (light) part of the score package suitcase icon.
- Select the icon of class **Note**
- Choose File⟶Make Alias
- Drag the newly created alias to the **user** package class hierarchy window.

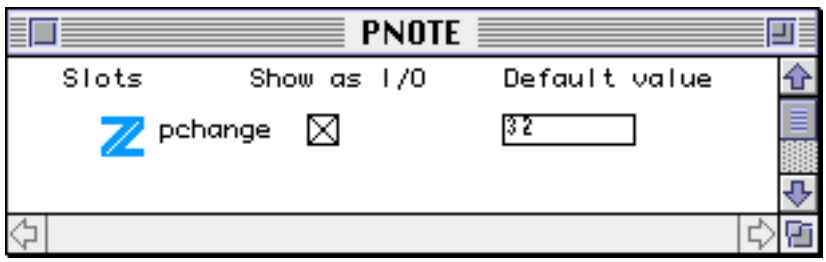
- Connect **Note** to **Pnote**.



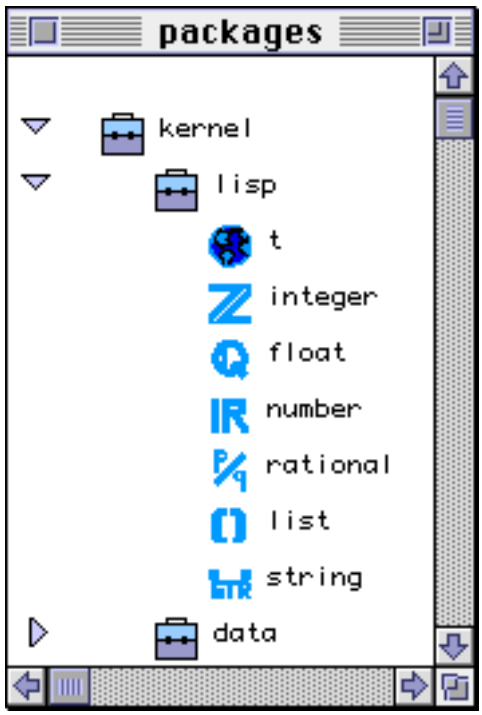
Adding slots to classes

Now, we add a slot *PChange* to the class **Pnote**. We'll use this slot to store the value of the program change.

- Double-click on the icon of the class **Pnote**. A class slots editor window appears.

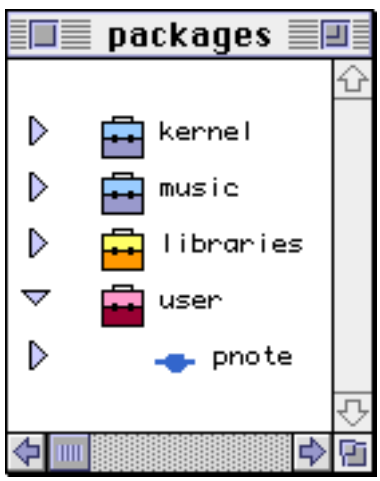


- Choose File⟶New Slot
- Locate the icon of the class **integer** from the package **kernel:lisp**.

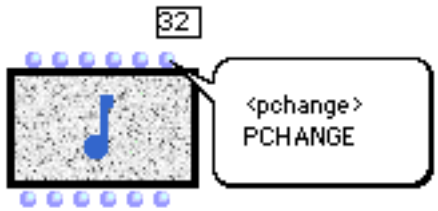


- Drag the icon of the class **integer** from the package **kernel:lisp** to the icon defining the class of the slot in the class slots editor window .
- Enter a default value for the slot.
- Check the Show check box. When checked, the slot will be visible as an input to the **Pnote** factory. Otherwise, the slot is hidden.

The class has been created. Check the **user** package:



Drag it to a patch window in order to get a **Pnote** factory.

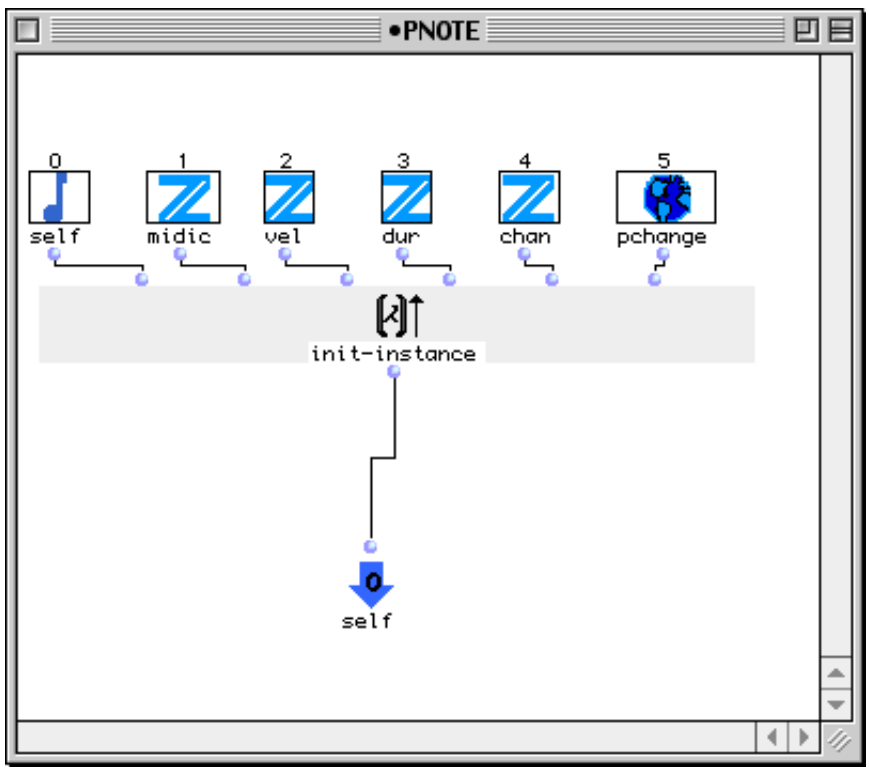


Notice that all the slots from **Note** are present in addition to our new slot, *pchange*, with its default value. Note also that **Pnote** has inherited a graphic editor from **Note**.

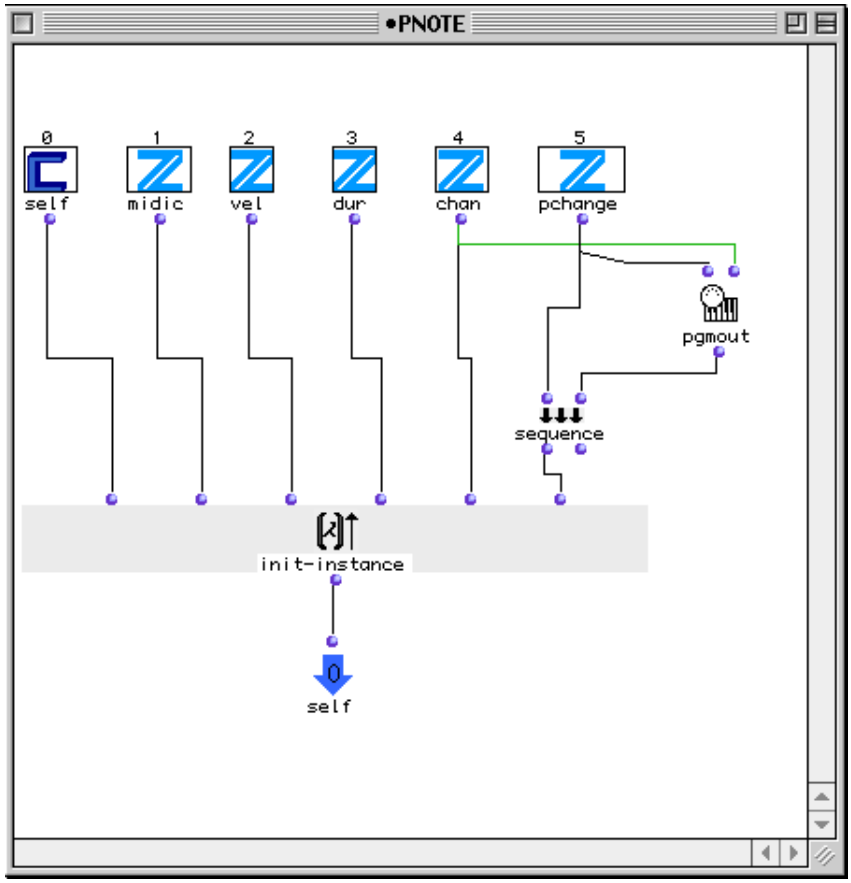
Changing the initialization method for a class

That takes care of configuring the new slot for the program change. But all that does is hold a value. How do we get it to actually issue the program change? It turns out a factory can be programmed to process the data in its slots, both before and after creating the instance of the class.

If you option-double-click on the **Pnote** factory icon, an instance initialization window appears where you can establish a mapping between the inputs of the factory and the inner slots of the instance to be created at initialization time, i.e. when the instance is created, i.e. evaluation.



You may insert functions, patches, etc. between the input icons (i.e. *midic*, *vel*, etc.) and the inputs of the `init-instance` box. In order for our program change to play on the correct channel, we must add a `pgmout` function, with input from the appropriate slots:



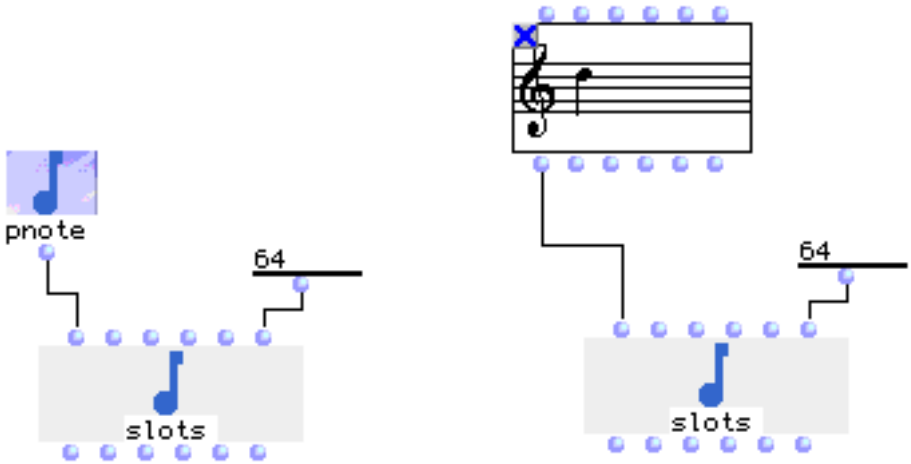
Note the use of the `sequence` function. Though it sends the program change anyway, `pgmout` returns nil at its output when it is evaluated. This nil would thus be output at the *pchange* output of our **Pnote**. We want the value of the original slot to pass, so we use `sequence`, which evaluates all of its inputs but only returns the output evaluated, which in this case is the value of the slot *pchange*.

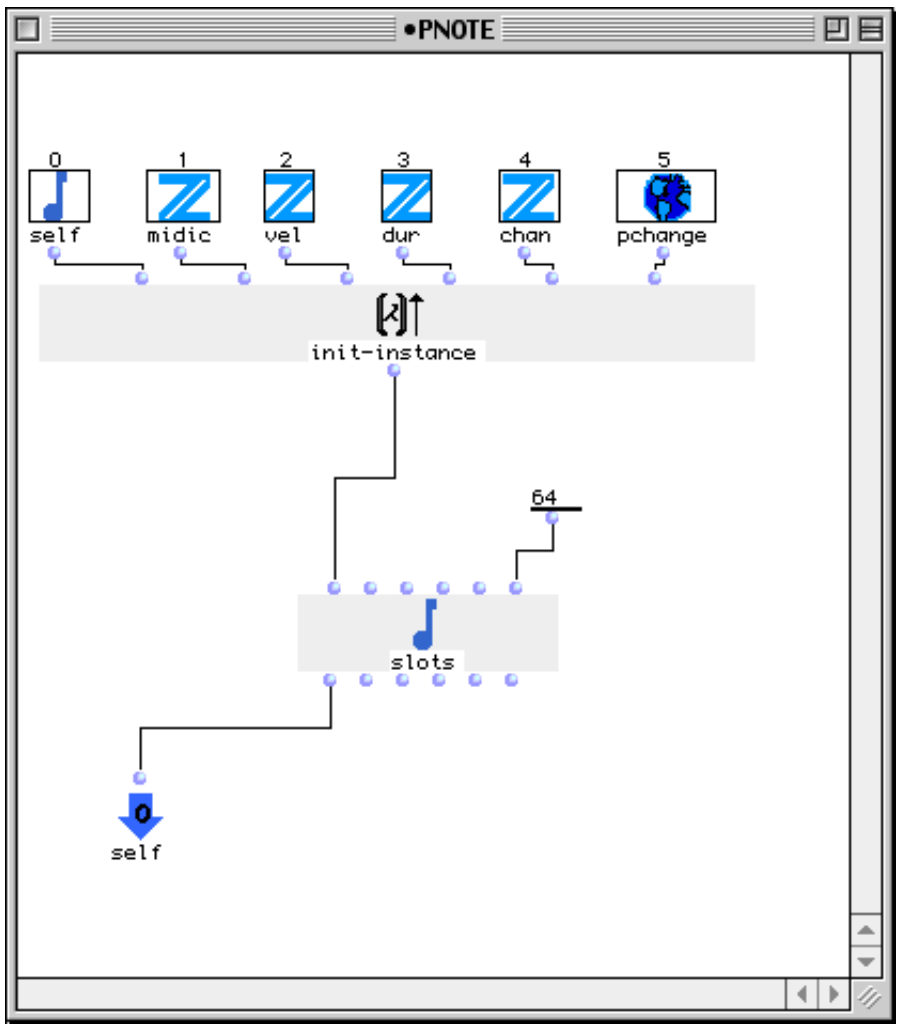
The newly-created instance is available at the single output of the `init-instance` box. You may modify it 'after the fact' using the "get-set-slot" construction.

The get-set-slot construction

When you drag a class icon from a package to a patch window, you create a Factory. If you **shift**-drag the same class icon (or if you **shift**-drag the factory icon within the patch window), you get a get-set-slot icon specialized for that class, labeled `slots`.

The Get-set-slot, or `slots`, is a function that takes an instance of the class from which it was created at its *`self`* input. This can be a factory *`self`* output, the instance coming from `init-instance`, or a variable (instance created with option-**shift**-click). If you evaluate the leftmost output of the get-set-slot, you get the object connected to its leftmost input, unchanged. If you evaluate the other outputs, you get the values of the slots of the object connected to the leftmost input. Here's where it's different: If you connect something to its other inputs, you change the value of that slot *within the object connected to the leftmost input.* Any of these examples would set the *pchange* slot of the **Pnote** to a value of 64:

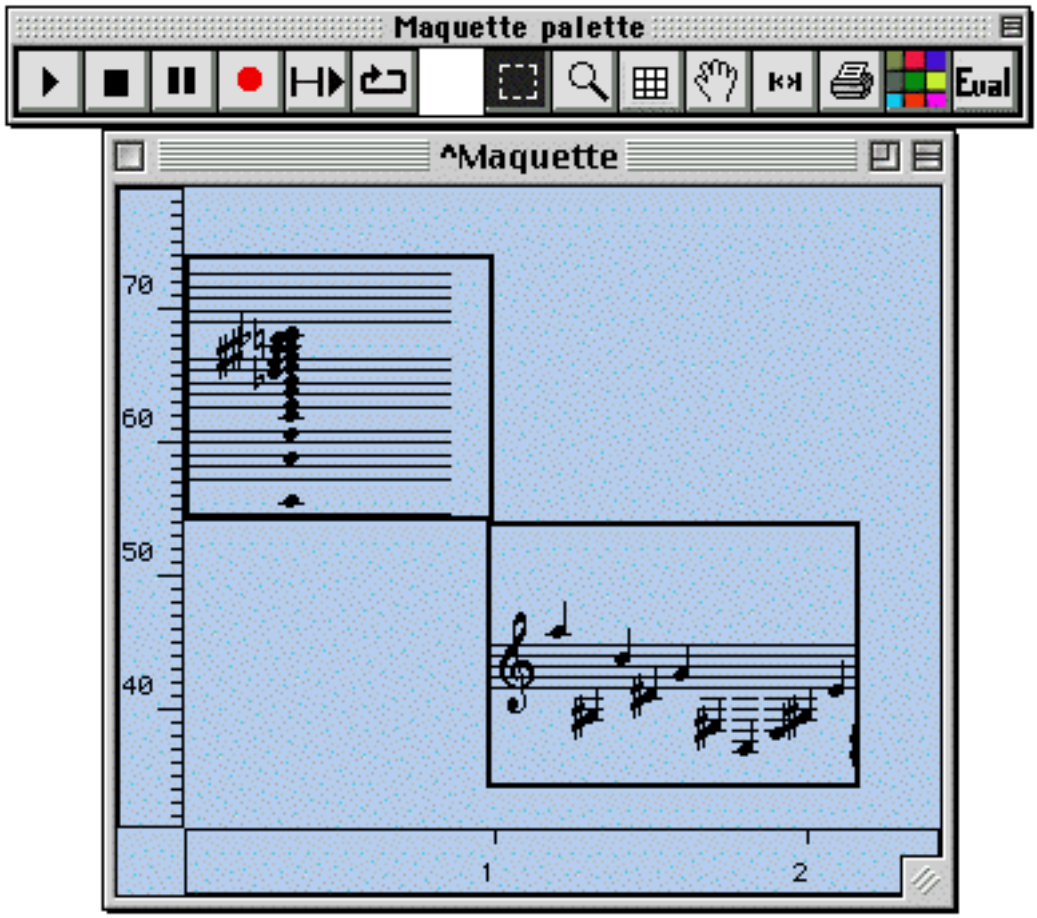

pnote
64
slots
64
slots


•PNOTE
0
self
1
midic
2
vel
3
dur
4
chan
5
pchange
init-instance
64
slots
0
self

Chapter 10. Maquettes

Note: For those curious, *maquette* is a French word which translates as *model* or *sketch*.

The **Maquette** is a special container object which incorporates a time element. A **Maquette** can contain any musical object, but can also contain a special object called **Temporalbox**. These boxes can contain other patches and even other **Maquette**s.



Any OM playable musical object may be put in a **Maquette**, as may MIDI files, AIFF files, other **Maquette**s. When these objects are dropped into a **Maquette** they are represented by boxes in the frame. Double-clicking the box opens a graphic editor for the object associated with it. You may also drop whole patches into a **Maquette**, in which case they are automatically enclosed in a special object called a TEMPORALBOX, described below. Objects in the **Maquette** are placed in time, along the x-axis, and are played in that order. In addition, if the object is a patch, (within a **Temporalbox**) it has access to information about its placement and shape within the frame of the **Maquette**, and these quantities can thus be used to change the way the patch behaves. This information comes into the **Temporalbox** through the outputs of the special *self* object. In addition, you can create your own inputs and outputs between **Temporalbox**es which can carry other data between them.

Finally, the output of a **Maquette** may be saved directly as a midi file using the `save-as-midi` function.

The structure of a Maquette

Objects in the Maquette frame

Acceptable types

A **Maquette** object may contain any number of the following objects:

- An AIFF sound file
- A MIDI file
- A patch
- Another **Maquette**
- Any playable (i.e. score class) factory
- A class icon directly from a package window if that class is available for use in **Maquette**s (same restrictions as for factories).
- A variable created inside a patch (with shift-option-click). Variables must be produced from one of the objects listed above.

Adding objects to the Maquette

Objects can be added to the **Maquette** in four different ways:

- Any of the object types may be dragged right into the **Maquette** frame from the Finder or the Workspace or the `packages` folder.
- An empty **Temporalbox** can be created by command-dragging a box in the **Maquette** frame when the Select tool is active.
- Objects can be added to the **Maquette** via the inputs to the **Maquette** icon when used in a patch window:

The second input of the **Maquette** box takes a list which can contain any of the above objects, for insertion into the **Maquette**. The first input takes a list of onsets, measured from the zero point, in milliseconds, which determine where along the time axis the objects at the second input will be placed. If the onsets list does not have the same number of elements as the list of objects, then the difference of the last two onsets is used as the distance between remaining objects.

- Using the function `addbox2maquette` (described in the reference and demonstrated in the tutorials section) and the **Maquette** box in reference mode.

The Maquette frame

The axes of a **Maquette** are numbered. The x-axis is a time dimension, labelled in seconds. The y-axis is an arbitrary value which can be used to control the **Temporalbox** objects in various ways. You can change the scale of the axes by dragging their numeric labels. The **Temporalbox**es and other objects themselves are represented by colored boxes in the field. They can be resized by moving the cursor over their lower right corner and dragging (as with any OM box).

The contents of the boxes can be view directly through the minivisualization (just like in the patch window), which is turned on by selecting a **Temporalbox** and hitting **m**. Double-clicking on the boxes opens the associated editor or the patch window, if the box is a patch. To jump directly to the musical object connected to the output of that patch from the frame window, Hold option while double-clicking.

The General Palette

Play button	Plays the **Maquette**. Evaluate first with the Eval button.
Stop button	Stops playback.
Pause button	Pauses playback.
Record button	Currently disabled.
Play Selection button	Plays back only the boxes currently selected, or the time span selected using the **z** command followed by dragging in the frame.
Loop Playback button	Causes the play sequence to repeat indefinitely when selected.
Select tool	The default tool, used for selecting boxes by clicking or dragging rectangles to enclose them, and for dragging around the boxes themselves.
Zoom tool	Adjusts the axes so that a rectangle dragged with this tool is enlarged to fill the frame.
Display Grid toggle	Turns display of gridlines on and off grid.
Hand tool	Allows you to adjust the view by dragging in the frame.
Recenter View button	Recenters the view and adjusts the axes so that all boxes fit in the frame. Also resets the scale of the metric ruler (see below).

	Print button	Currently disabled.
	Color Picker button	Brings up the Mac color picker so you can choose a color for the selected box. If no box is selected, sets the color of the background.
	Eval button	Evaluates all the boxes in the **Maquette**, preparing them for playback. You must click this after making changes and before playing in order for the changes to be heard.

Contextual menus

Holding ctrl while clicking on **Maquette** boxes or the background of the frame brings up a contextual menu with relevant commands. When used on a box, they are:

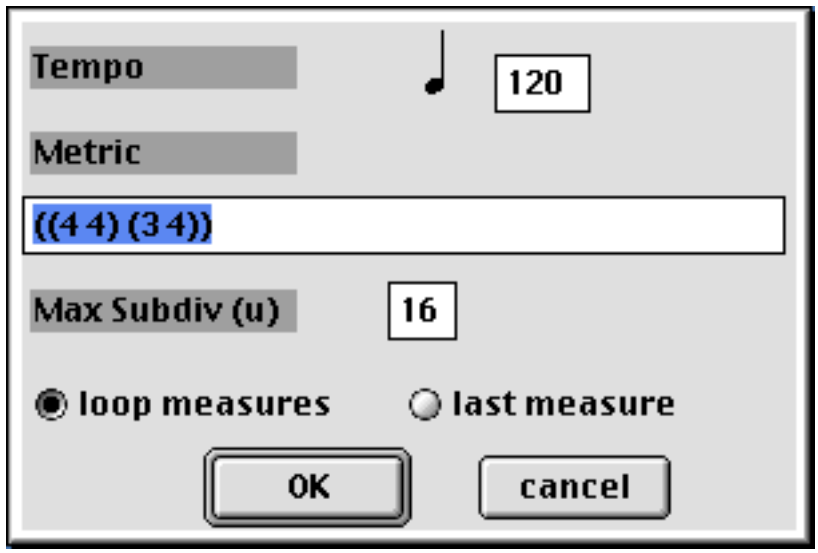
- Open - opens the object
- Get info - opens an inspector window (equivalent to selecting the object and hitting command-**i**) where you can adjust certain attributes of the box. Changes are confirmed simply by closing the box.
- Midi Ports - opens a dialog where you can set the input and output MIDI ports of the object, overriding settings made for objects within the box.

When brought up on the background of the frame, the contextual menu contains the following elements:

- Eval - equivalent to the Eval button.
- Play - equivalent to the Play button.
- Hide (Show) connections - turns on and off display of patch cables between **Maquette** boxes.
- Hide (Show) metric ruler - turns on and off display of the metric ruler (see below.)
- Last Saved - reverts to the last saved state. Useful since there is no Undo function in OM.
- Midi Ports - opens a dialog where you can set the input and output MIDI ports of the entire **Maquette**, overriding settings made for objects within it.

The Metric Ruler

When turned on with the contextual menu option described above, a third ruler bar is shown along the top of the frame. It displays a series of measures, by default in 4/4. The small tick marks are beats and the numbered tick marks are bar numbers. This ruler is automatically calibrated to agree with the absolute time ruler in the bottom of the frame, according to the tempo set in the dialog below. Dragging to change the time scale at the bottom of the frame also adjusts the scale of the metric ruler. However, the time display is not updated when the metric ruler is dragged. (This is a graphic bug only- dragging the time scale axis a bit will update it.) To fix this, click the Recenter button on the tool bar. To adjust the metric ruler, double-click it, which brings up the metric ruler settings dialog:



Changes to the tempo value will be reflected in the relationship between the metric ruler (top of frame) and the absolute time bar (bottom of frame) when the dialog is closed.

The Metric option allows you to change the time signatures of the measures displayed in the ruler. They are entered as a list of lists in the form (A B), as in a rhythm tree.

When the metric ruler is turned on, the horizontal position of all objects is constrained to an invisible grid, and changes in their position are 'snapped' to this grid. The Max Subdiv field lets you set the resolution of the grid. It is measured in subdivisions of a 4/4 bar, regardless of the time signatures of the metric ruler. The default value is 16, i.e. sixteenth notes. Setting this to 4, for example, constrains all object positions and movement to quarter notes against the metric bar.

The loop measures/last measure radio buttons control how the Metric field is interpreted. When set to loop, the entire sequence is repeated *ad infinitum*. When set to last measure, only the last measure in the list is repeated.

Markers

Markers are used to align objects in maquette precisely. You add a marker by **option**-clicking in the x-axis at the bottom of the maquette frame. A little flag appears representing the marker. You can link boxes within the frame to this marker by holding **shift** and dragging a connection from the marker to a box. If you drag the connection to the front (left) end of the box, the start time of the box is 'snapped' to the marker, that is, the `offset` of the box is changed to match the marker's position. If you drag to the back (right) end of the box, the end of the box is 'snapped' to the marker's position, that is, the `stretch-fact` is set such that the box ends at the marker's position. A line indicates the connection. You may connect as many boxes as you want to a marker, after which moving the marker moves all of them simultaneously. To delete a connection or a marker, click the line representing it to select it and hit **delete**. You may need to move the marker a little bit in order to select it before deletion. When the marker is selected you may hit Command-**i** to view its attributes.

The Temporalbox object

When a patch is dropped into a **Maquette**, it is enclosed in a **Temporalbox**, which has two unique elements: the *self* box and the *tempout* output. The temporal box returns information about the geometry and positioning of the box within the frame. You can use these values to affect the behavior of the patch. The output *tempout* must be connected to a playable music object, as this is the object that will be played within the **Maquette** when the play pointer crosses the left edge of the box. Additional inputs and outputs may be added with the usual buttons as with any patch. These are represented in the **Maquette** frame as tick marks on the upper and lower edges of the box, and may be used to create connections between **Maquette** boxes in order to pass data between them.

The *self* outputs

The *self* object represents the box itself. The 10 outlets return the following information about the box in the context of the **Maquette** frame:

- *self* - The **Temporalbox** object itself. Can be used with addbox2maquette and related functions.
- *offset* - The distance of the left edge of the box from the zero point of the x axis, in milliseconds.
- *extend* - The duration of the object (or of the musical object connected to *tempout*, if a patch) as played, not as represented graphically, in milliseconds. Changing the scale of the view will not affect this value. If you have changed the size of the box, this real duration of the object is nonetheless represented graphically; the extension of the box beyond the real duration will not be colored like the front portion of the box.
- *colorframe* - A value representing the color assigned to the box.
- *value* - The object connected to the box's *tempout*
- *posy* - The height of the top edge of the box as measured against the y axis.
- *stretch-factor* - The ratio of the distance spanned by the box graphically against the time axis to the real duration of the object. A value of 1 means the object is drawn at 'actual length'. Values smaller than 1 represent a compression, larger values an expansion.
- *sizey* - The length of the left edge of the box as measured against the y axis.
- *free-store* - An open slot where you may store values using the get-set-slot mechanism. See the chapter on Classes for more information on the get-set-slot mechanism.
- *Reference* - The OM internal reference for the **Temporalbox** object. The *self* object can't be put in reference mode, so you must take this value here if you need it. See the Chapter on Patches for more info on reference mode.

Maquettes in patches

Maquettes can also be manipulated from within patches by using their inputs and outputs rather than opening the frame and dragging objects in. See Tutorials 32, 33, and 40 for more information.

The functions addbox2maquette, removetemporalbox, temporalboxes, and get-maquette allow you to perform operations on the **Maquette** via its Reference mode. See their Function Reference entries for more information.

Chapter 11. Midi and OpenMusic

Introduction to MIDI in OpenMusic

MIDI (Musical Instrument Digital Interface) is a music industry standard communication protocol that enables MIDI instruments and sequencers (or computers)to communicate. It can be considered as a set of musical commands messages which electronic instruments send to control each other.

In OpenMusic, the musical objects are played (or recorded) with MIDI using the MidiShare library. In a basic configuration, MIDI messages sent by OpenMusic (via MidiShare) are recieved and played by the default computer built-in MIDI device (QuickTime). However, other MIDI instrument (like synthesisers, expanders, samplers,...) can be connected to the application (see Midishare documentation in ports setup section).

The OpenMusic MIDI toolkit will allow you to control directly the communication between OpenMusic and MIDI instruments, with a set of tools for analysis, filtering, conversions and control of MIDI data.

All MIDI objects and functions are to be found in the Music/Midi menu.

Basic concepts

MIDI messages

All MIDI communication is based on messages called events. When a sequence of notes is played, it's first internally converted into a sequence of MIDI events. These MIDI events are then sent to the MIDI instrument to be interpreted. So it's important to understand what a MIDI Event message is made of. The MidiShare library provides a high level set of attributes used to define MIDI events. These attributes are the values we will deal with while working with MIDI :

a) Date :

Is the date when the MIDI event is supposed to be sent.

In OpenMusic's default configuration, units for dates are miliseconds. (1000 units per quarter note, tempo 60 bpm = 1 quarter note / second)

b) Port :

Each **MidiEvent** is sent to a particular port (port 0 by default), where it will be recieved by the MIDI device(s) connected to this port. (see Midishare documentation in ports setup section)

c) Reference number :

The reference number can be used to determine a device reference ID, but it's generally used in Midi files to determine the differents song tracks (usually one track per instrument).

Note: Track 0 is generally used for Tempo and metric informations (events of type Tempo, TimeSignature,...) which can not be applyed to a single track.

d) Channel :

MIDI provides 16 different channels in standard. All MIDI events are adressed to one single channel (1 - 16). However some particular events (called "Meta-events") doesn't affect only one channel; in this case channel number is ignored.

e) Event type :

The event type is an identifier for the type of MIDI message.

Annex 1 is a table of all MidiShare events types

f) Fields :

The value(s) sent for the event. Number of fields depends on the event type. For example, for a "KeyOn" event, values sent are pitch and velocity, etc.

The annex table describe fields of each MIDI events type.

About MIDI events

In this section we will explain some of the most important features regarding MIDI events.

Note events

We call Note Events Midishare events of type KeyOn, KeyOff, and Note

(the type Note event is internally converted into a KeyOn event followed by a keyOff event after a corresponding delay according to the Note duration).

In fact, while converting MIDI files or sequences in OpenMusic, it is these kind of events that are dealt with (For voice and poly, we also consider tempo and time signature MIDI events). In multi-seq or poly conversion, track number of the event determine the voice in which the notes will be displayed.

Control Change events

The Control Change events (type CtrlChange) set values for the instrument channel controls. The first field of a Control Change event is a control number which specify which controller we want to set. A list of standard controllers numbers is provided (see Selection Tools). The second field of the event is the controller's value setting (0 - 127).

There can be continuous controllers, on/off switch controllers, data controllers, or undefined controllers. Not all controllers are implemented by transmitting devices. On/off switch controllers are considered as Off for value < 64 and On for value > 64. Some Continuous controllers (number 0 to 31) accept fine settings, with 7 bits of additionnal definition (called LSB value) in an extra controller (number 32 to 63).

Program Change Events

Program Change events (type ProgChange) set a specific instrumental timbre or sound for the channel it's adressed to. The General MIDI standard bank of 128 timbres and sound can be used to choose the desired sound. (see Selection tools)

`Pitchbend` events

Pitch Bend events (type `Pitchbend` or `Pitchwheel`) set the Pitch Bend Controller of a MIDI Channel. This control has 14 bits resolution (-8192 to 8191). However, OpenMusic MIDI tools provide using both 7 bits (-64 to 63) or 14 bits resolution.

The sensitivity (pitch range) of the Pitch Bend depends on the MIDI recieving device. In most cases, it's a 2 tone range.

e.g. : If pitch range = 2 tones (-1 to +1), with 7 bits resolution (-64 to 63), value 32 represent a 1 semi-tone upper bending.

For using microtones in openMusic, We use Pitch Bend settings to detune some midichannels. Micro tonals notes will then be dispached in the corresponding micro-tuned channels. See details in OM Tutorial Playing MIDI.

Using MIDI in OpenMusic

Notes can be considered as MIDI objects.

In Notes, Chords, Chord-seq, and other score editors, each note has its own MIDI channel. That means that the event sent for playing this note will be sent to this channel, to which can correspond any particular setting (instrumental timbre, `volume`, effects, `pitchbend`,...). Changing the note's channel is equivalent to change the MIDI event's channel.

The same thing exists for MIDI ports. The MIDI port determines thru which port the events will be sent. Then the MIDI devices connected to this port will recieve the MIDI message. It's an important setting to consider if you want to play with one or more external MIDI devices. The MIDI port can also be set for each Note.

Score editors also have a record function, with a InPort parameter. This one determines the port from which will be sent messages for MIDI recording. Eventual remote keyboard must be connected to this port.

See MidiShare port setup for MIDI ports configuration in MidiShare's own documentation.

Objects conversions :

Objects containing compatible MIDI informations can be processed with MIDI we'll see in this documentation the use of a special method `get-midievents` which extracts MIDI events from them.

MIDI and Musical objects (**MidiFile**, **EventMidi-seq**, Chord-seq, Voice, ...) can be converted into one another, provided they contain some compatible data, by simply connecting the "self" output slot of an object with the "self" input slot of another.

Sending MIDI events with simple functions

Some MIDI functions send MIDI events when they are evaluated. Such evaluations will modify MIDI playing device state. One of the input is a MIDI channel or list of channels on which the event will be sent. Other optionnal input is MIDI port. If no port is specified, the event will be sent to the default MIDI port.

- `midi-o` : General output function. Sends input to the serial modem port.
- `pitchbend`/`pitchwheel` : Send `Pitchbend` events (Change tuning of MIDI instrument)

  Pitchweel use 14 bits precision (-8192 to 8191)

  `Pitchbend` use 7 bits precision (-64 to 63)

  Used to set micro tones : see OpenMusic tutorial "Playing MIDI".

- `ctrlchg` : Sends control change events. Parameters of a Control Change event are a controller number and a value (0-127). For example (10 64) stands for "set controller 10 (pan) to 64 (middle)".
- `volume` : Sends `volume` event with a `volume` value.

  Note: `volume` is a particular Control Change event (controller number 7)

- `pgmout` : Sends Program Change events with a program number (0 - 127)

  Used to set instrumental timbre to be applyed to different MIDI channels.

  (see OpenMusic tutorial "Playing MIDI").

- `aftertouch` : sends a channel pressure event with pressure value.
- `polykeypres` : sends a key pressure event with pressure and pitch values.
- `sysex` : Sends a system exclusive message.
- `midi-reset` : Resets default MIDI settings

  Note: Selection Tools (section 4) can make the use of these functions easier...

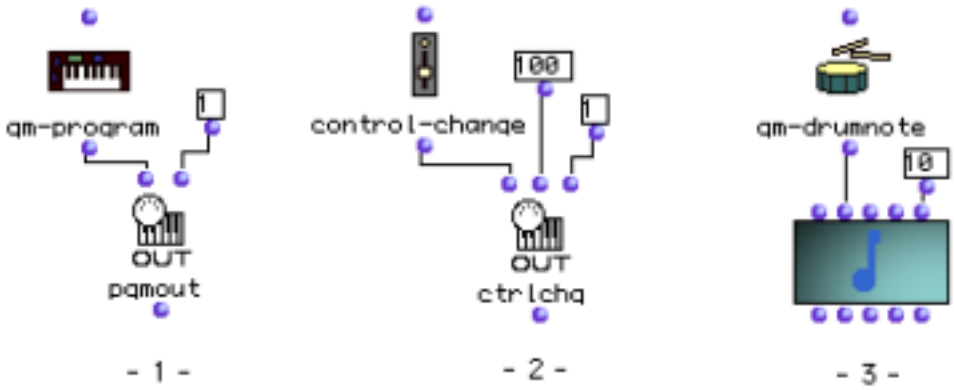
(see the patch "01-midievents" MIDI " included in the folder "midi tutorial" in "OMWorkspace")

Selection tools

For easier access to Midi objects and functions, some "menu-functions" allow to choose parameters with their standard names internally converted into corresponding integer values.

- `gm-program` : select a MIDI program (General MIDI specifications)
- `gm-drumnote` : selects the note corresponding to a drum element (General MIDI)
- `control-change` : select a control change number
- `ms-event` : select a MidiShare Event type number

Examples :



(1) : Select the instrument timber to set to a MIDI channel.

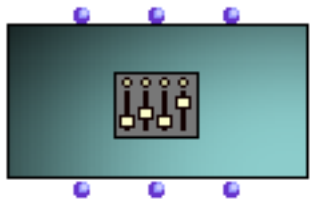
(2) : Select the controller number for a Control Change message.

(3) : Select the pitch of a Note supposed to play a drum element (channel 10 is usually reserved for drums.)

We will see further on the practical parametric capabilities of these tools regarding objects and functions settings.

The Settings Controller

The Settings-Ctrl class is a visual MIDI controller for MIDI settings.



Input slots determine :

- MIDI Outport
- Number of tracks in the console table

In this console, the main MIDI controllers can be set. For each track, a channel, a MIDI program (using General MIDI standard specifications), pan, `volume`, pitch bend, and two extra choosable controllers could be set.

Settings are sent by pressing the "space" key while the controller is active.

The Settings-Ctrl can also be considered as a MIDI object. It can be put in a maquette to initialize or change MIDI settings.

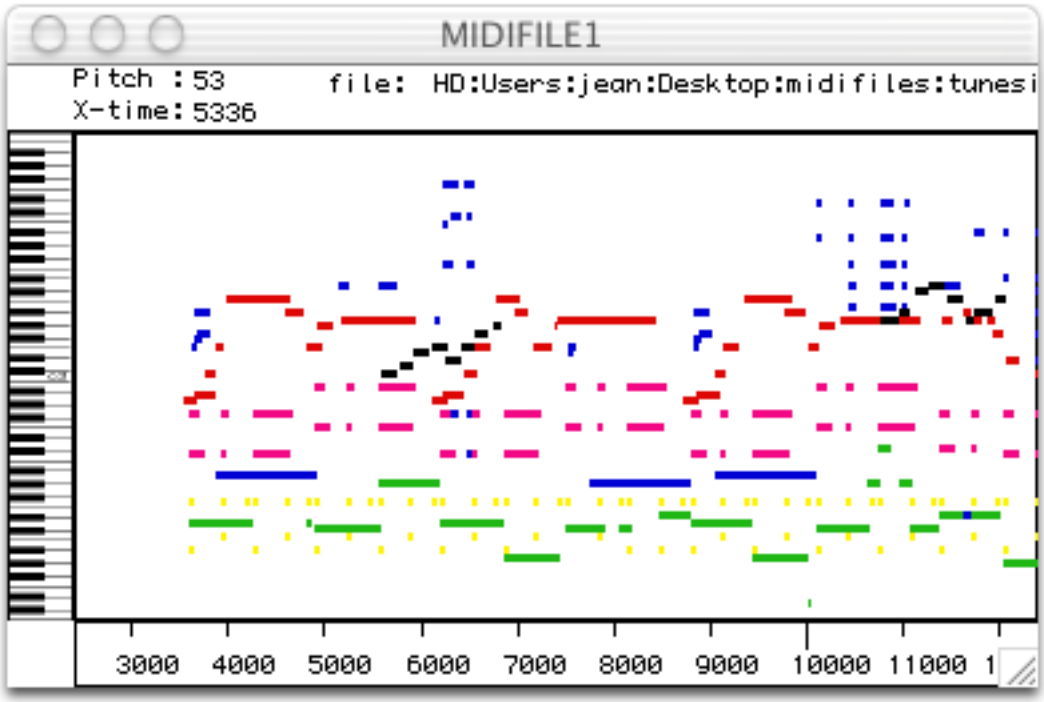
MIDI Files

The MidiFile box

The **MidiFile** is a file containing a sequence of MIDI events.

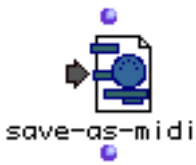
Evaluating the box will open a file-chooser dialog to select a MIDI file to open.

The **MidiFile** box have a specific MIDI Editor :

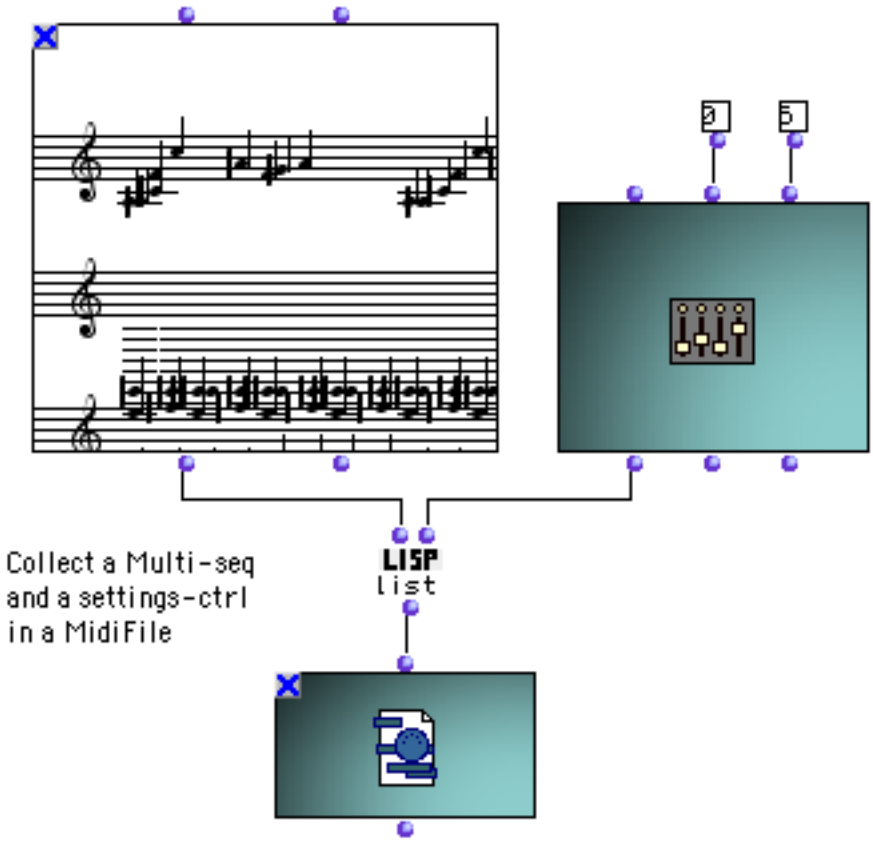


In this editor, each MIDI track is represented with a different color.

Saving OpenMusic objects in MIDI files



All Musical and MIDI objects (including maquettes) can be stored in a **MidiFile**, using the `save-as-midi` method, or by connecting them to the **MidiFile** box input.



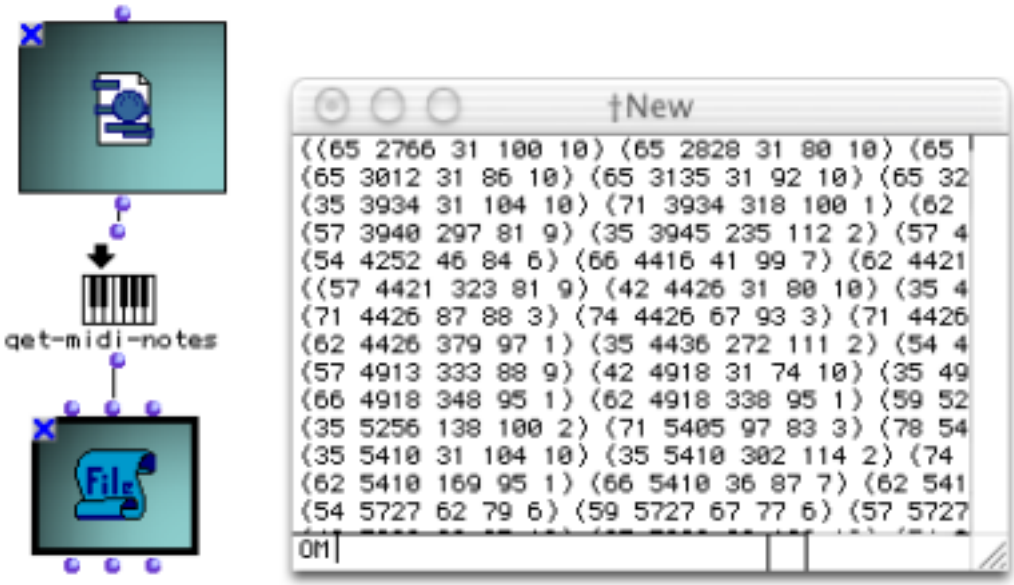
Saving a multi-seq and a settings-ctrl in a midi file

MidiFile informations

MIDI events processing can be done by using the `get-midievents` method (see **MidiEvent**s section). However, some special methods output specific informations from **MidiFile**s :

- Notes

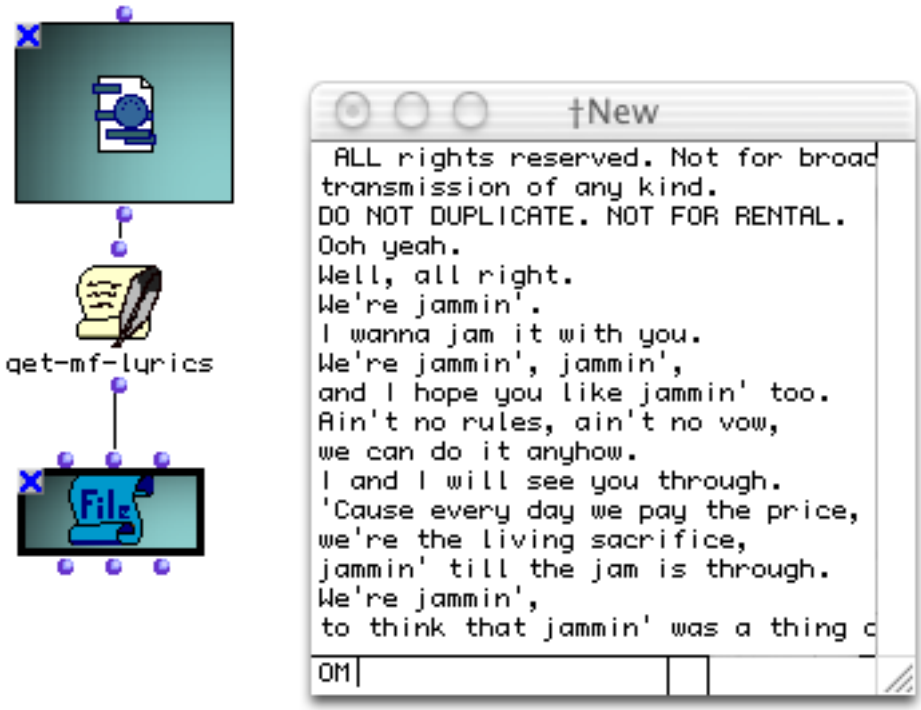
The `get-midi-notes` method returns a matrix of notes. Each list of the matrix is a track containing notes. A note is a list (pitch offset duration velocity channel).



MidiFiles can be coneverted into musical sequences or superpositions (chord-seq, multi-seq,...), which is another way to extract note informations from **MidiFile**s.

- Lyrics

Some MIDI files contain events of type Lyrics in which we can read eventual lyrics of the file song. The `get-mf-lyrics` extracts all these lyrics into a single string.


get-mf-lyrics
File
†New
ALL rights reserved. Not for broad
transmission of any kind.
DO NOT DUPLICATE. NOT FOR RENTAL.
Ooh yeah.
Well, all right.
We're jammin'.
I wanna jam it with you.
We're jammin', jammin',
and I hope you like jammin' too.
Ain't no rules, ain't no vow,
we can do it anyhow.
I and I will see you through.
'Cause every day we pay the price,
we're the living sacrifice,
jammin' till the jam is through.
We're jammin',
to think that jammin' was a thing c
OM

MIDI Events

The MidiEvent box

The **MidiEvent** box represents a single MIDI event in OpenMusic.

As explained in section 2, **MidiEvent** class slots are: type, date, track, port, channel, and fields.

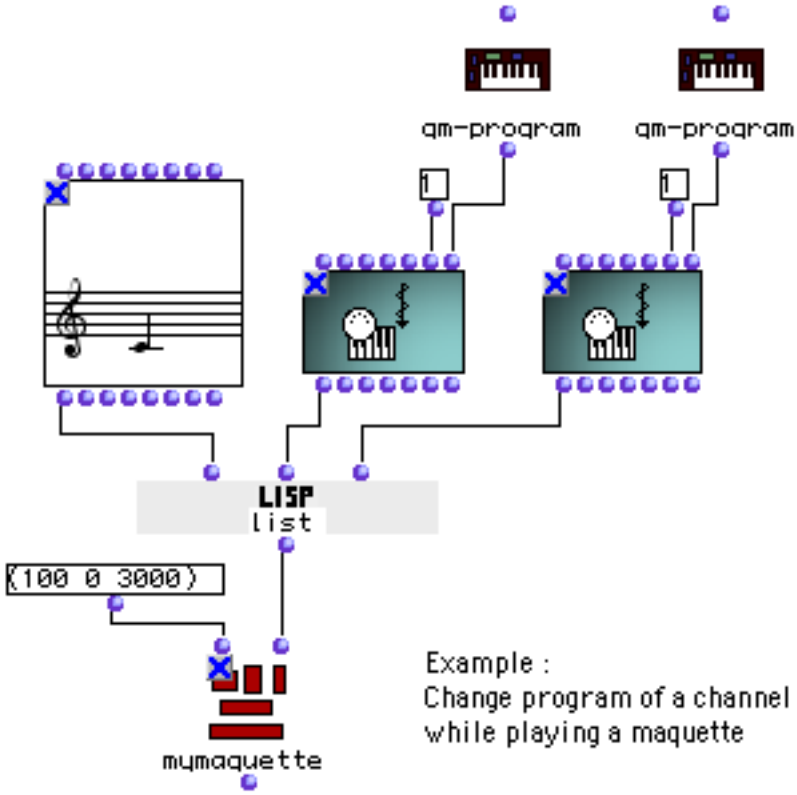
In-popup-menus and selection tools (see Selection Tools) can help to set **MidiEvent** inputs slots.

A **MidiEvent** can be played (sent to MIDI outport) by pressing "P" key while its box is selected.

A **MidiEvent** can also be used, like any other MIDI (or musical) object in a maquette.

Note: If a date is assigned to a **MidiEvent**, it will be sent in milliseconds at a corresponding time after play is selected.

Example :



A chord-seq put in a maquette with two **MidiEvent**s for changing MIDI program

Processing MidiEvents

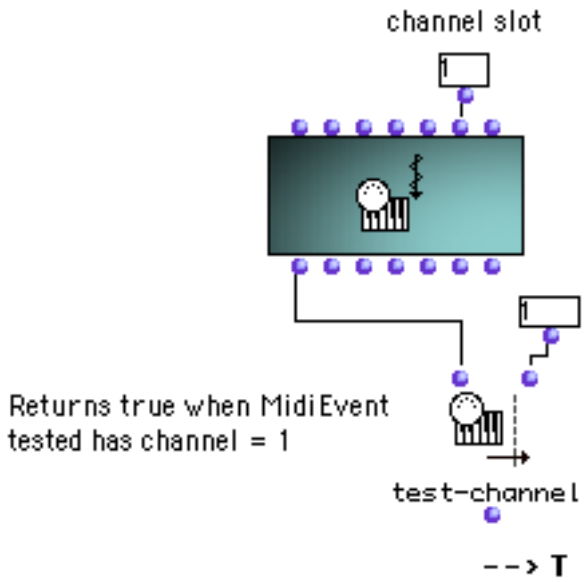
Mostly all types of MIDI or musical objects (chord-seq, voice, note, eventmidi-seq, midifile, settings-ctrl,...) can be converted into a list of **MidiEvent**s using the method `get-midievents`. It's a very useful function, since it allows all kinds of conversions and "MIDI processing" of objects.

This function has an optional input in which can be connected a lambda test function for **MidiEvent**s. Using this test, **MidiEvent**s extracted from an object can be filtered.

Here are some functions that can be used to filter **MidiEvent**s :

- `test-date` : tests if **MidiEvent**s occur between a min date and a max date.
- `test-type` : tests the **MidiEvent** type.
- `test-track` : tests the **MidiEvent** track.
- `test-port` : tests the **MidiEvent** port.
- `test-channel` : tests the **MidiEvent** channel.
- `midievent-filter` : various tests on **MidiEvent** slots (type, track, port, channel).

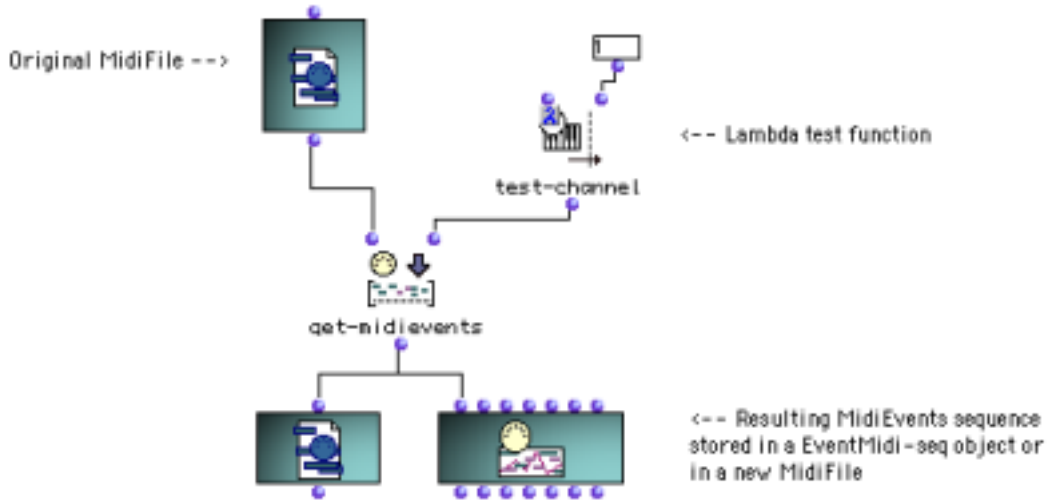
Example :



A test on **MidiEvent**'s channel

Now let's use these test functions to filter MIDI events extracted from a container with `get-midievents`.

In this example, we are applying a filter to all **MidiEvent**s from a **MidiFile** with the same test as in the previous example :

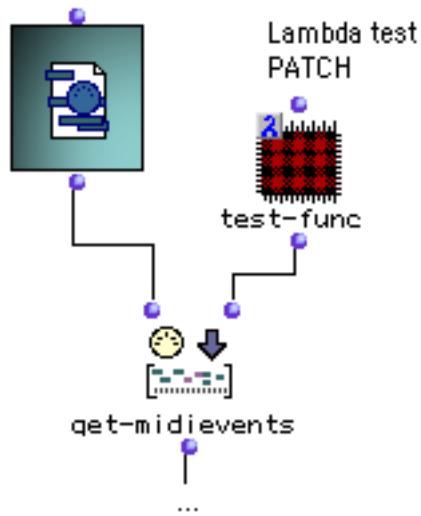


`get-midievents` outputs a list containing all **MidiEvent**s which channel is equal to 1.

The test function is a parameter of get-midievent, therefore it must be a lambda function and have only one free input (for the **MidiEvent**s to be tested).

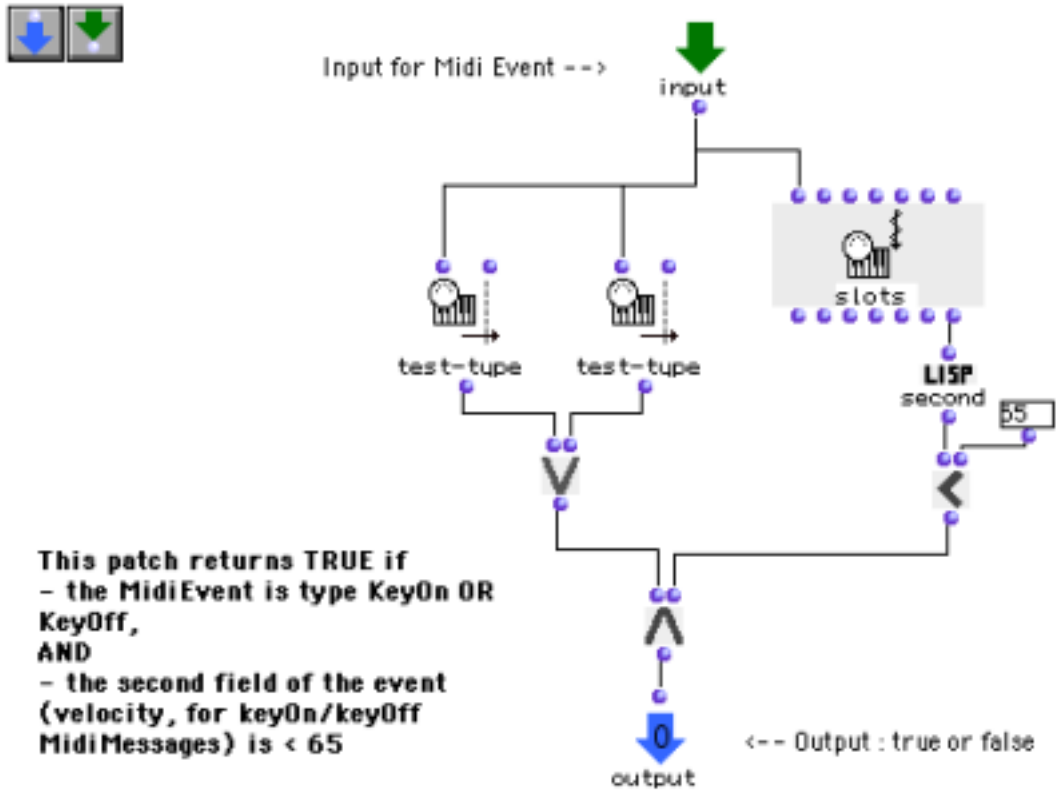
The output list can be saved as a new **MidiFile**, or stored in a **EventMidi-seq** object.

For complex filtering functions, a patch (set as a lambda function, with a single free input) can also be plugged to the `get-midievents` test input for complex filtering functions, a patch:



MidiEvents of a **MidiFile** filtered with a test-patch

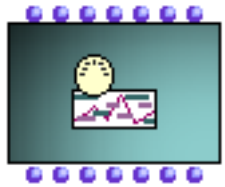
This is an example of what could be inside the test patch (extract from tutorial patch filters) :



Note: Some MIDI events types are called "textual" (typeSeqName, typeInstrumentName, type-Lyrics, typeCopyright, ...). It means that their content (fields) is a textual coded information. To translate these fields into text, use the `me-textinfo` method. It can be applyed to **MidiEvent**s or **MidiEvent**s lists.

MIDI sequences

The EventMidi-seq box



The **EventMidi-seq** object represents a sequence of MIDI events. Its slots are lists of the corresponding **MidiEvent** slots (types, dates, tracks, ports, channels, fields).

It can store all types of **MidiEvent**s (notes, controllers, etc..) and in this way can be compared to a **MidiFile** object. The main difference is that **EventMidi-seq** is not written on your hard disk as a file, so it can be manipulated more easily.

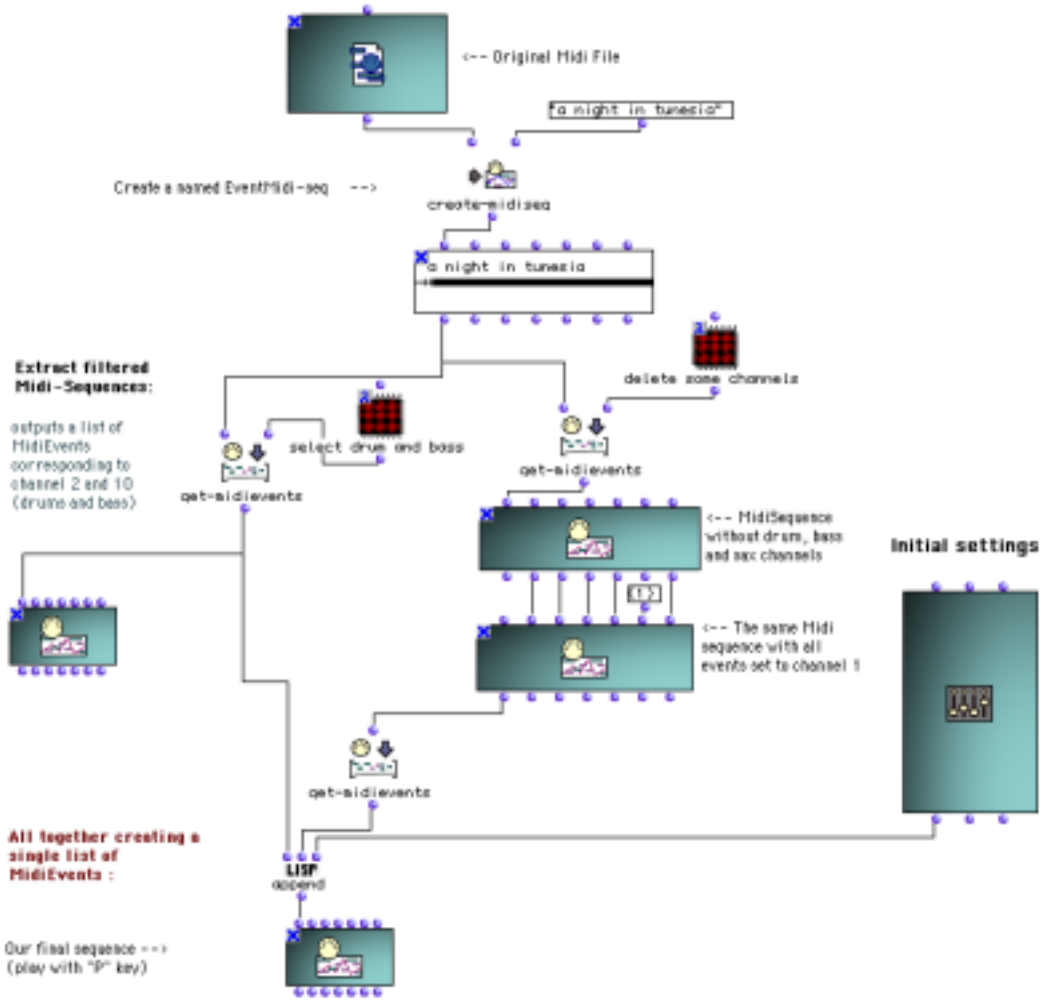
An **EventMidi-seq** can be constructed by setting its input slots or by connecting its self input with a list of **MidiEvent**s. So, as we have already seen regarding all MIDI and musical objects that can be converted into a list of **MidiEvent**s (using `get-midievents` method), all these objects can also be directly converted using an **EventMidi-seq** or a list of objects.

The **MidiEvent**s can be extracted and filtered from **EventMidi-seq** with the `get-midievents` method. The method `get-midi-notes` (see **MidiFile** section) can also be applyed to **EventMidi-seq**.

EventMidi-seq will be usefull for temporary storage of filtered or processed MIDI data.

It can also be played in maquettes, or by using the "P" key in a patch.

The following example is extracted from midi sequences tutorial patch :



In this example, **MidiEvent**s are extracted from a **MidiFile**, and then filtered to separate channels in order to aplly different process. The resulting sequences are put in a new **EventMidi-seq** with a setting-ctrl object to add the initial settings events to the sequence.

Note: Using the function `create-midiseq` converts a list or an object into an **EventMidi-seq**. The second (optionnal) input is for setting a name to the newly created sequence. This name will appear in the **EventMidi-seq**'s miniview.

Events are automatically time-sorted in each converted **EventMidi-seq**. However, manually created sequences can be sorted using the `temporal-sort` method.

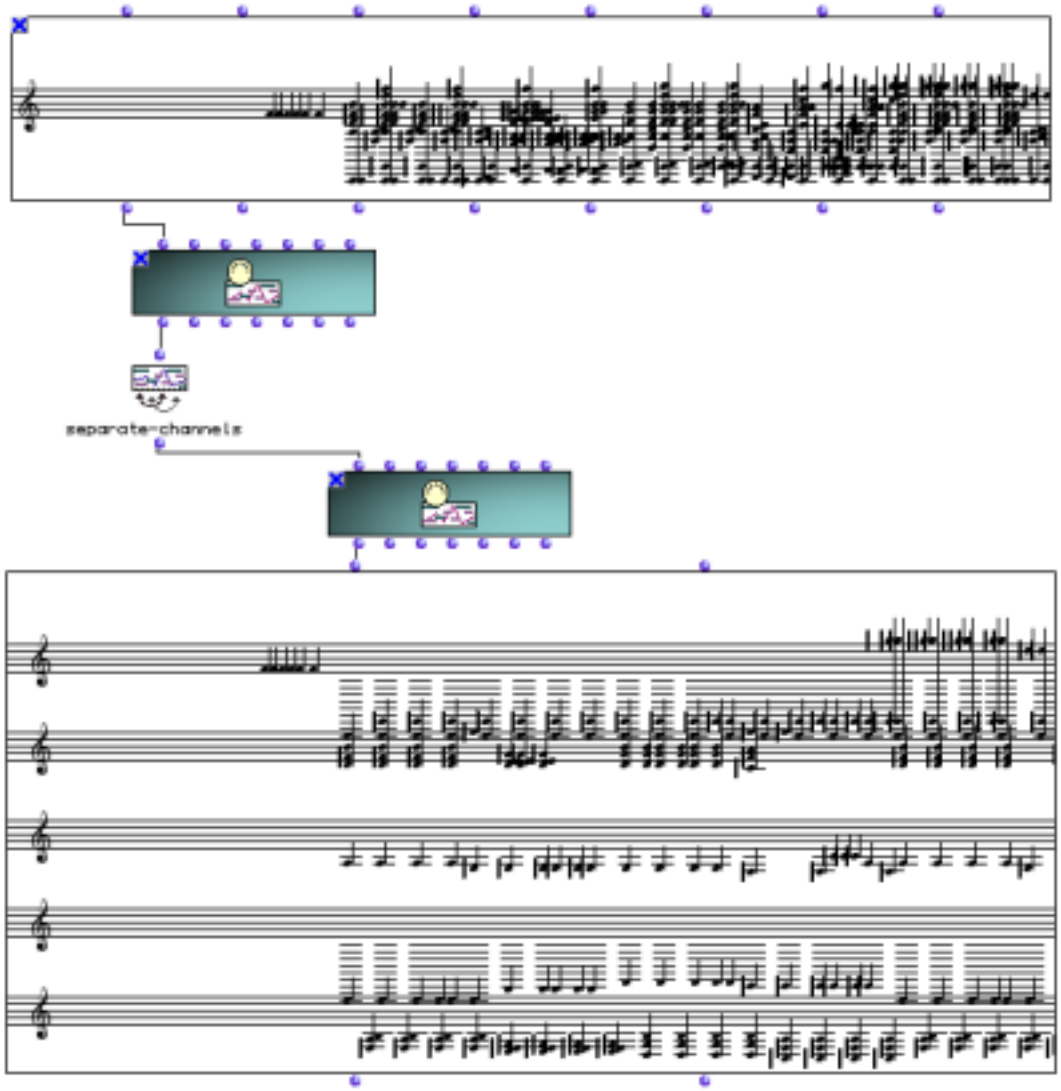
Using Midi sequences for musical objects operations

All OpenMusic musical objects can be converted into MIDI sequences. Note informations (and tempo, measure markers for voice and poly objects) are converted into MIDI events.

This conversion allows special treatments (filters with `get-midievents` method, ...) and operations on **MidiEvent**s.

The function separate-channels processes all Events of the **EventMidi-seq** and put each channel on a diferent track.

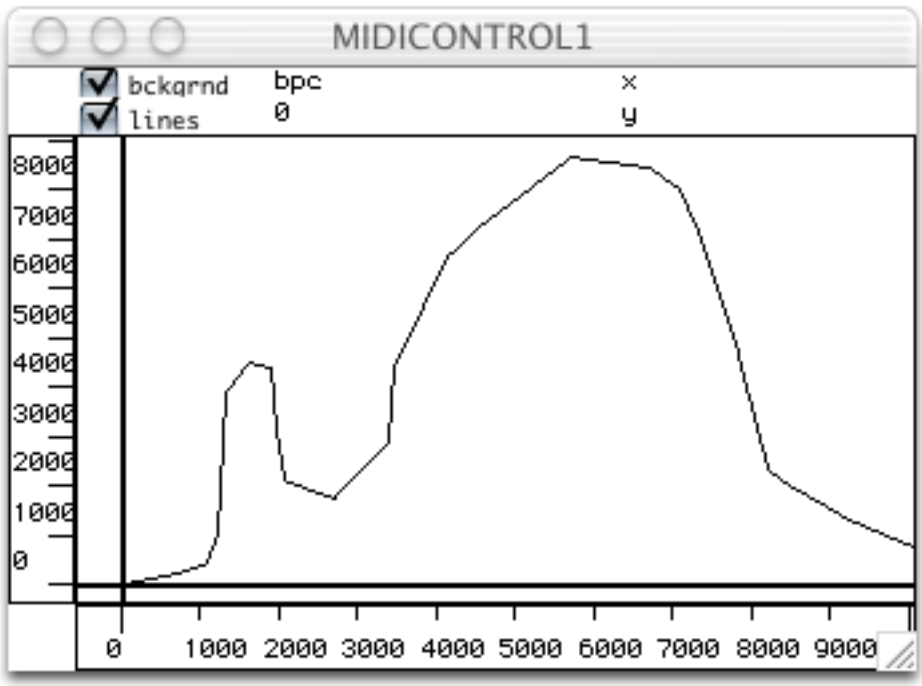
In MIDI sequence to Multi-seq, or Poly conversions, a voice is created for each MIDI track. In the following example, we separate notes of different channels in chord-seq into different tracks via **MidiEvent**s processing :



MIDI Continuous Controllers

The MidiControl box

MidiControl is a special case of Midi Sequence representing a single MIDI continuous controller evolution. Such controller can be `Pitchbend`, or any kind of Control Change. It contains a single event type, a single channel, port and track, and a list of dates and values. In this way it's like a kind of BPF object set to control a MIDI parameter. In reality, channels, ports and tracks can be lists of numbers. In this case the same continuous controller (dates/values) applies to all the channels/ports/tracks of the lists. Dates/values BPF can be edited with the BPF **MidiControl** editor :

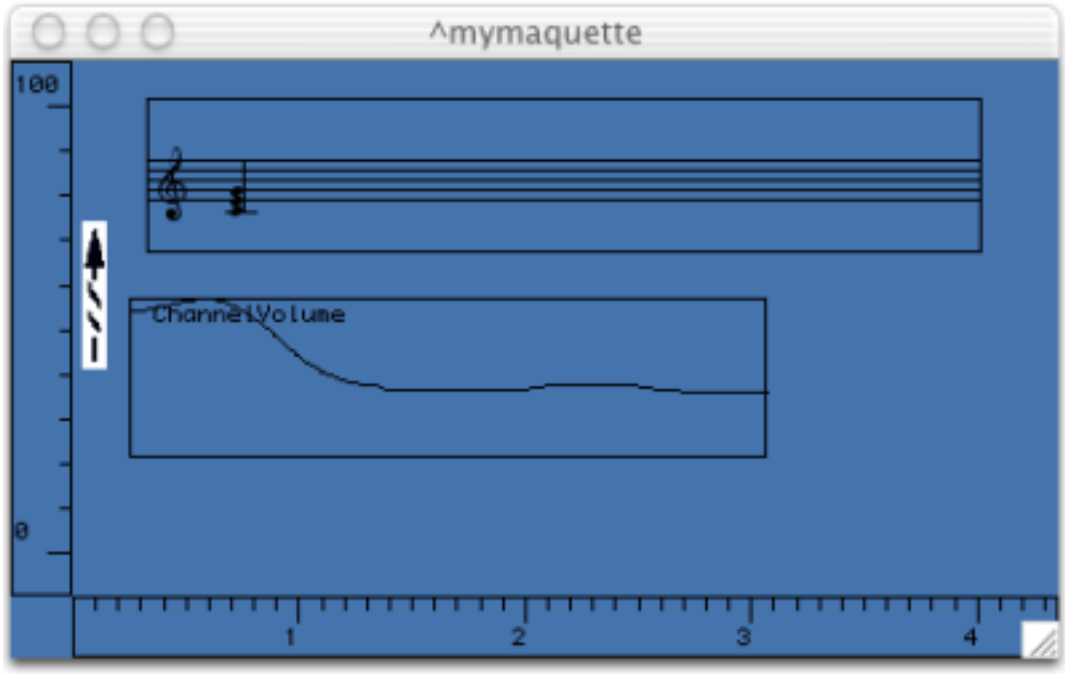


MidiControl editor

Like all other MIDI objects, it can be converted (and filtered) in a list of **MidiEvent** (`get-midievents` method), and in **EventMidi-seq**.

The **MidiControl** can be played by pressing "P" key when the box is selected. It can also be put in a maquette. There it can be streched to fit the desired duration.

The following example shows a Channel `Volume` controller in a maquette :



More examples of **MidiControl** application in a continuous controller tutrorial

Continuous controllers values

A List of controllers is available as an input menu for the first input (controller type) of the **MidiControl** box.

Most of the controllers are classical ControlChange controllers. For this type of controllers values must be given in 0 - 127 range (7 bits). (For On/Off switch controllers, value < 64 means "off", and value >= 64 means "on").

Some of them (controllers 0 to 31) have a corresponding "Fine" controller (32 to 63). For these controllers values must be set in 0 - 16384 range (14 bits).

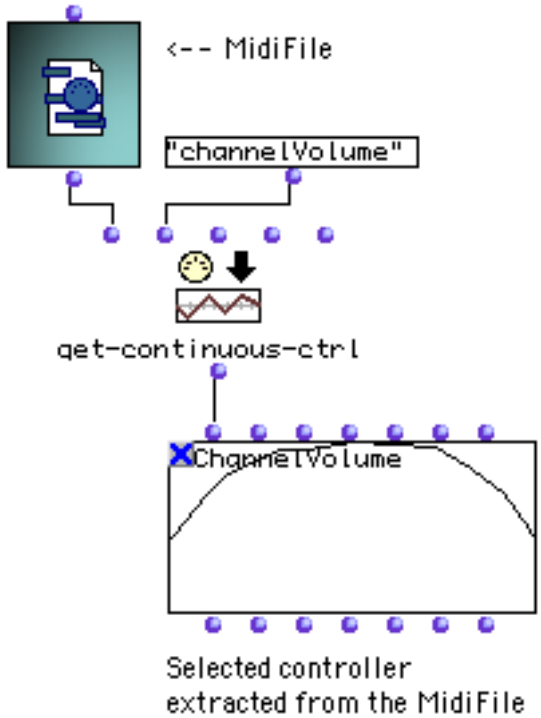
The `Pitchbend` controller's value ranges from -64 to 63, and -8192 to 8191 for "fine" `pitchbend` (also called `pitchwheel`).

Creating MidiControls

A **MidiControl** object can be created using its input slots, but also by diferent ways :

- Using its BPF editor
- Extracting a Midi continuous controller from another Midi object

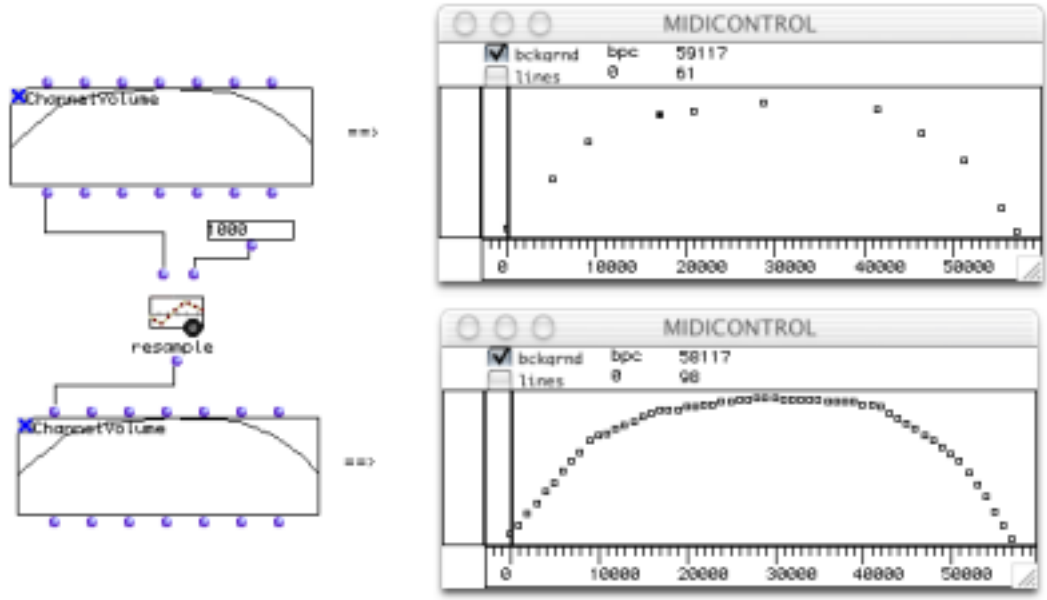
A **MidiControl** object can be extracted from Midifile or **EventMidi-seq** using the `get-continuous-ctrl` method.



Channel `Volume` controller extracted from a **MidiFile**

- Resampling an other Midi Controller

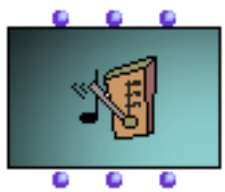
The **MidiControl** BPF (dates/values) can be `resampled` using `resample` method. This function creates a new control BPF with a regular given sample rate.



In this example, the initial controller is `resampled` with a 1000ms sample rate.

Tempo and Measures

The Tempo-Map box



The **Tempo-Map** object allows separation of tempo and metric information for works on quantification and analysis.

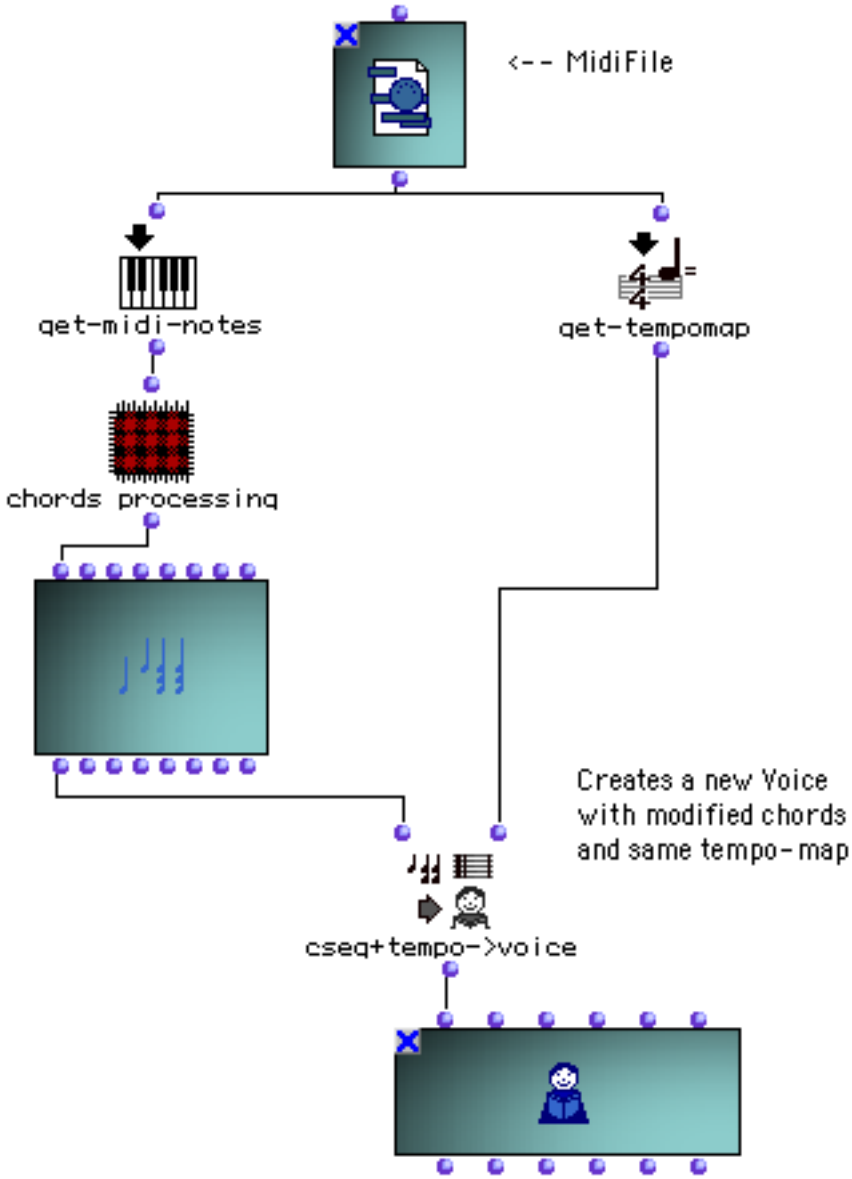
It contains a list of tempo changes (date , tempo), and a list of measure markers (date, MIDI time signature).

Tempo-Map can be extracted from any type of musical or MIDI containers (provided it contains tempo or metric information : **MidiFile**, **EventMidi-seq**, Voice, Poly, Measure) using `get-tempomap` method.

Tempo-Map can be processed like other MIDI objects (conversions, `get-midievents`,...).

Improving quantification

With **Tempo-Map**, Chord-seq to Voice conversion can be improved with `cseq+tempo->voice` method. The tempo and measure informations will be used to quantify chords.



In this example, notes and tempo map are extracted from a midiFile (could be from a voice, for example). Notes are processed as chord-seq and then re-written in a voice using the initial tempo and metric informations.

Tempo-Map is also internally used to convert **MidiFile** to voice or poly considering tempo and measure when they're included in the **MidiFile**.

Saving objects as **MidiFile** will also save tempo information and measure markers as **MidiEvent**s in the created **MidiFile**.

However, tempo information is used with some limits :

- a Midi file cannot contain different tempo maps for each track :

  if a poly with different tempo is saved, all tracks are set to tempo 60.

- Voices and polys don’t (yet) accept tempo changes.

  If a **MidiFile** with variable tempo is converted to voice, only first tempo will be taken in account.

MidiShare MIDI events table

Event Number	Event Name	Event fields with value ranges	Description
0	Note	• **Pitch**, note number (0 -127) • **Vel**, velocity (0 -127) • **Dur** duration (0 - 215-1)	A note with pitch, velocity and duration.*When a Note event is sent to external Midi devices, actually a NoteOn message is first sent followed, after a delay specified by the duration, by a NoteOn with a velocity of 0 to end the note*
1	KeyOn	• **Pitch**, note number (0 -127) • **Vel**, a note velocity (0 - 127)	Note On message with pitch and velocity.
2	KeyOff	• **Pitch**, note number (0 -127) • **Vel**, a note velocity (0 - 127)	Note Off message with pitch and velocity
3	KeyPress	• **Pitch**, note number (0 -127) • **Press**, key pressure (0 - 127)	Polyphonic Key Pressure message with pitch and pressure

Event Number	Event Name	Event fields with value ranges	Description
4	CtrlChange	• **control number** (0 -127) • **control value** (0 -127)	Control Change message with controller and value
5	ProgChange	• **program number** (0 -127)	Program Change message with a program number. *(used to set the instrument for a MIDI channel).*
6	ChanPress	• **channel pressure** (0 -127)	A Channel pressure message with pressure value
7	PitchWheel PitchBend	• **LS 7-Bits of 14-bits pitch swing**, (0 -127) • **MS 7-Bits of 14-bits pitch swing,** (0 -127)	Pitch Bender message with a 14 bits resolution. *In OM, also accepts a single value in (-8192 to 8191) range*
8	SongPos	• **LS 7-Bits of 14-bits location,** (0 -127) • **MS 7-Bits of 14-bits location,**(0 -127)	A Song Position Pointer message with a 14 bit location (unit : 6 Midi Clocks).
9	SongSel	• **song number** (0 -127)	Song Select message with a song number.
10	Clock	-	Real Time Clock message
11	Start	-	Real Time Start message
12	Continue	-	Real Time Continue message
13	Stop	-	Real Time Stop message
14	Tune	-	Tune message
15	ActiveSens	-	A Real Time ActiveSens message
16	Reset	-	Real Time Reset message

Chapter 12. Preferences

The Preferences pane is brought up by selecting File⟶Preferences from anywhere within OM. When you change settings, be sure to click OK or else they will not become valid. If you want to keep the Preferences set for the next session, click Save.

There are four subpanels: Genreal, Midishare, Notation, and Csound, represented by icons along the left border of the Pane. Their options are described below:

The General Preferences

Special file writer and Message Error handle buttons

These are settings for MCL's interaction with OM. You should not have reason to adjust them.

Comment Style button

Brings up a dialog box where you set the default font, size, and style for any commment objects you add to your patches. These settings can be changed individually for each comment via the contextual menu.

Comment Color button

Brings up a dialog box (the Apple color picker) where you set the default color for any commment objects you add to your patches. These settings can be changed individually for each comment via the contextual menu.

User Name field

Your name here.

Screamer Valuation buttons

Controls the manner in which values are assigned to the screamer variables of the backtrack module.

The Midishare Preferences

Here you can set the default input and output Midishare port for newly created objects only. These settings will not replace those already set for existing objects. They can be overridden by settings made within individuals objects after creation.

The Midishare SetUp button takes you to the **msdrivers** control application:

Configuring and installing Midishare® is covered in the Getting Started section.

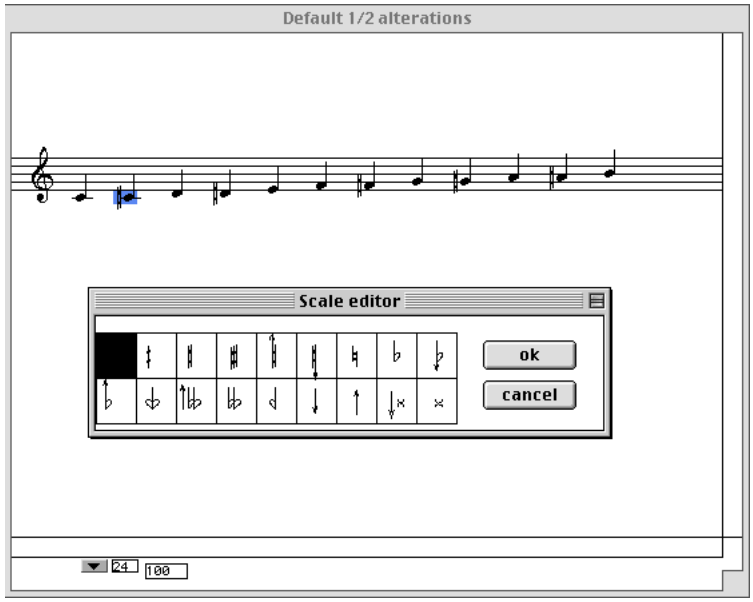
The Notation Preferences

This panel controls the default appearance and behavior of musical objects in OM.

Enharmonic spellings

The Approx radio buttons control the default display mode for 'rounding' of midic values in the graphic editors for music objects. You may choose to display objects in semitones, quartertones, or eighthtones by selecting the appropraite button.

Double-clicking any of the Approx buttons brings up an editor window where you can select the way enharmonic equivalents are displayed for that display mode. By default, OM displays all chromatic pitches as raised notes. You can choose an alternate spelling for each note by selecting the note and hitting **o**, which brings upa list of alteration symbols to attach to the note:



Changing the alteration changes the symbol displayed in front of that note; you must still change the scale tone by selecting the note and using the arrow keys. For example, to change all A♯s to B♭s, you would select A♯ in the scale editor hit **o**, and click the ♭ icon, click

OK, which results in the A♯ changing to an A♭. Then, with the A♭ selected, hit the ↑ key to change it to a B♭.

Music Font Size field

Sets the default font size in the music editors. Set by dragging the value itself up and down with the mouse.

Delta Chords field

Colors buttons

These two buttons bring up the Mac Color Picker window where you can select a color for the display of the staves and notes in musical objects.

Staff System menu

This sets the default staff system choice for newly created music objects. 'F' and 'G' represent bass and trebl clefs, respectively. Doubled letters indicate an additional cleff of that type displaying notes in octave transposition up or down (depending the clef), useful if your objects have lots of ledger lines.

Quantify button

This button brings upa dialog box where you make the default settings for the omquantify method. This method is used to transcribe durations into traditional musical notation, and is used whenever you connect the *self* output of a **Chord-seq** object to the *self* input of a **Voice** object.

These settings are identical to those of the stand-alone function `omquantify`. See the reference entry on `omquantify` for a complete explanation.

Dynamic Table

This brings up the Dynamic editor. When you select the dyn option from the display options popup menu in a musical editor, traditional dynamic symbols are displayed next to each not, based on the value of the *vel* (key velocity) slot for that note, a value between 0 and 127. The Dynamic editor lets you set which dynamic symbols correspond to what range of velocity values. OM automatically adjusts the other ranges so that there is no overlap.

The Csound Preferences

These preferences apply to the omChroma package. They are discussed in the library documentation.

Chapter 13. Libraries

A library is a specialized set of class and method definitions written directly in *LISP* (as opposed to a class or method that you create with OM's graphic interface).

Libraries add extra functionality to OM. Many libraries are included with OM on the CD-ROM, and should have been automatically installed simultaneously with it. They are stored in the `UserLibrary` folderm within the OM folder itself.

You can browse the available libraries in the packages window in the special libraries subpackage (the yellow suitcases). In order to use the library, you must first load it by double-clicking the lower half of the suitcase icon, after which the suitcase opens and you can view the contents. The functions and classes of the suitcases will also become available in the Functions and Classes menu bar when in the patch window.

OM 4.6 Glossary of Terms

A

Atom

A discrete unit of data in a list. Although lists may be nested, an atom is **never** a list. It is a single piece of data. Atoms would include the number 3, the fraction 1/4, and the string "hello".

B

Boolean Logical Operators

A system of functions developed by the English mathematician George Boole (1815-64). Boolean logical operators are based on comparisons of truth values, which can be either true or false. In computers, these are usually represented by 0 or 1. LISP has a special truth value, t to represent true, with nil representing false. Boolean operators compare two or more truth values and return a truth value. Two basic examples are `AND` and `OR`. `AND` compares any number of truth values and returns true if they are all true. `OR` compares any number and returns true if any one among them is true. See the functions `omor` and `omand`, for example.

C

car

The first element of a list. For example, the `car` of (1 2 3) is 1. This rather cryptic designation has its roots in ye olde days of the IBM 704, an old mainframe that ran one of the first versions of LISP. Pieces of data were stored with two fields, an "address" and a "decrement". The address was the first part of the register, the decrement was the second. The decrement could point to some other address register, and in this way a chain of elements could be created. `car` originally stood for **C**ontents of **A**ddress part of **R**egister, i.e. whatever came first. `cdr` stood for **C**ontents of **D**ecrement part of **R**egister, i.e. whatever came after.

In Common LISP, `car` came to mean the first element of a list, and `cdr` the rest of the elements. If the `car` of (1 2 3) is 1, the `cdr` of (1 2 3) is (2 3). By extension, the `cadr` is the `car` of the `cdr`, i.e. the first element of the rest of the list, i.e. the second element. The `caddr` is the first element of the second part of the second element (are you still with me?), i.e. the third element.

Mercifully, LISP includes plain english functions `first`, `second`, and `third` so our heads don’t explode.

Cardinality

The number of pitch classes in a pitch set. (0 3 7) is said to have cardinality three.

cadr

See: car

caddr

See: car

cdr

See: car

cddr

See: car

Class

A Class is a very common concept in object-oriented programming languages. In OpenMusic, data structures (anything you store in order to keep track of) are described using classes. A class is a prototype for a data structure. For example, the class **Chord** is the prototype for all actual **Chord** objects. When you put a class icon in a patch window (by dragging or any other method,) you create a *factory* which refers to the class from which it originates. When evaluated, this factory generates objects. Classes are described in more detail in the Reference section.

Complement

The complement of a pitch set is the set of *pitch classes not* contained in it. Two pitch sets are complementary if the sum of their elements is 12 and they share no pitch classes. (They fill the octave with no overlap.)

D

Data Type

Type is an important concept in OM. Data types in OM include integers (numbers with no decimal point), rationals (fractions), lists, strings (lists of text characters like "hello"). Data type is important principally in determining the behavior of generic functions, which can deal with more than one data type. A data type is a class; therefore, **chord**s and **poly**s are also data types.

Certain functions expect certain data types. To imagine an analogy, answering "yes" to the question "how many apples would you like?" is confusing; this is because we expect a specific data type (a yes/no answer). Likewise, try using `om+` to add "hello" and

"my name is". You will get the error No method defined for inputs in box OM+. This is because om+ doesn't know what to do with text.

Driver

A piece of software which allows control of either a piece of hardware (you need to install a printer driver before you can use your printer with your Mac, for example) or another piece of software (Midishare requires a driver to communicate with the Mac's built-in QuickTime synthesizer).

E

Enigma Transportable File

An acronym for Enigma Transportable File. Enigma is the graphics engine used by the popular notation program Finale®. The ETF format is thus a type of "raw" file which can be imported into Finale. Finale can also produce ETF files of its documents using the File⟶Save As... function. OpenMusic can also read and produce ETF files, completing the bridge between the two programs. It is also worth noting that since the 'F' in ETF already stands for "file," the term "ETF file" is redundant in the same way as the oft-used expression "ATM machine."

F

Factory

A factory is a function that knows how to create *instances* of the class it is associated with. Factories are created by moving class icons to the workspace, and are represented by icons which you can evaluate, creating the instance.

Fibonacci Series

Discovered by mathematician Leonardo Fibonacci in the 12th century, a fibonacci series, also called a fibonacci sequence, is a series of numbers where each number is the sum of the two preceding numbers. It starts by definition with 0 and 1, giving 0+1=1 as the third term, 1+1=2 as the fourth term, 1+2=3 as the fifth term, etc. The first 10 terms of the sequence are thus: (0 1 1 2 3 5 8 13 21 34 55). The fibonacci sequence has the special property that the ratio between two consecutive terms approaches the golden section with increasing precision as the sequence progresses. The golden section is a proportion found in many naturally occuring patterns- the chambers of the conch shell, for example- and this proportion has been used by many cultures both ancient and modern in their art and architecture.[1]

1. http://evolutionoftruth.com/goldensection/goldsect.htm

Function

A function is a basic conceptual unit in OpenMusic. Functions are objects- they are represented by graphic icons on your screen. Along with classes, they make up the basic building blocks of OM. Generally, functions perform tasks, and classes are the materials on which these tasks are performed. This distinction is somewhat artificial. From a programming standpoint, at the most fundamental level, functions and classes are the same thing- objects. Therefore, functions can sometimes behave like classes and vice versa.

Functions in OpenMusic can be user-defined. Writing your own functions is covered in the User's Manual. Whether they are standard or user-defined, OM functions are generic.

Function Call

When you place an icon for a function in OM, you are graphically making what is known as a function call. To add two numbers, for example, you would use the function `om+`. You can think of the code for the function as the master copy of a book at the printing house. When you call the function, OM looks for the master copy of the code for the function within the body of OM code. A copy of this code is then made, like a copy of the book being printed for you to read. The function call is a like a bookmark, a reference to the code of the function, as if the entire code of the function were inserted. The `om+` function needs to know which numbers to add. These two numbers are passed to the function as **arguments**. In most cases, a function call will include arguments to pass along to the function so the function knows what to do. The arguments may be any kind of data, including other functions! Function calls can be placed within other functions. This is known as nesting. This is in effect what you are doing when you write *patches*. Your patch is itself a function which contains calls to other functions.

G

Generic function

All functions in OM are generic. Generic functions are also known as polymorphic functions. To understand generic functions, consider the following two situations. In the first, we want to add 1 and 2. In the second, we want to add 1 to all the elements of the list (1 2 3 4). In the first case we want to perform a simple mathematical calculation on the two quantities. In the second we want to apply that calculation to all the elements of the list. In OM, functions automatically choose the best operation based on the type of data they are presented with. This is the meaning of the designation 'generic.'

The specific procedure required in each case is called a method. A method is the set of instructions which tells a function how to deal with a specific type of data. Let's say we pass 1 and 2 to the function `om+`. It contains a method for dealing with two single numbers, which adds them together and returns the answer, 3. If we pass 1 and (1 2 3 4) to the function, a different method is called, a method which tells `om+` what to do

when the first input is a number and the second is a list. In this case, the method adds the number to all the elements of the list, and the answer, (2 3 4 5) is returned.

Generic functions are a very important ingredient of OM; they allow a large degree of flexibility and conceptual simplicity, since the same function performs many different tasks based on the type of data it is given. Since a data type is a class, we can imagine a function called `add` which, when passed two numbers, adds them, but which, when passed two chords, performs a union of their notes and returns a single chord.

Ghosted

Appearing in many contexts, ghosted items are visible but not available, either because they are in the background or because they are not valid choices. This is indicated visually by dimming or otherwise greying-out the button, icon, or object.

Graphic User Interface

A graphic user interface, also known as a GUI, is a program or part of a program which allows the user to visually manipulate information. The advantage of a GUI is that the user does not need to confront the raw computer code, and thus does not necessarily have to know anything about programming. Your internet browser is an example of a GUI (try selecting View⟶Source in Internet Explorer and you'll see the raw code of an internet page). OpenMusic is a GUI for the *LISP* programming language.

Group

In OM, a **Group** is a group of rhythmic values corresponding to a measure or part of a measure, independent of any pitch data. The **Group** is a class which may be combined with other **Group** objects into **Measures**. It is also a musical unit in the Notation Editors, being a group of notes beamed together or bracketed into a tuplet.

I

Instance

An instance is a single specific example of a *Class*, created by evaluating the*factory* icon. A group of specific notes in a **Chord** object is an instance of the class **Chord**, while the class itself simply defines what was stored (i.e., the notes).

inheritance

In an *object-oriented programming language*, inheritance is a property of *class* definitions. Classes may 'inherit' the *slots* and *methods* of another class, in addition to having their own slots and methods. This saves the time of defining all aspects of a class anew each time, and makes the program structure more logical since classes with related

functions are related in an inheritance tree. Creating a class which inherits from another is known as *subclassing* the parent class.

L

LISP

An acronym for **LIS**t **P**rocessing, LISP was created by John McCarthy in the late 1950's, making it one of the oldest programming langauges still in use today. LISP is an extremely flexible language; it allows users to make changes to code even while the program is running, and has the particularity that the structure of the code used to write functions and the structure of the code used to store data are the same, with the result that LISP programs can be made to write other LISP programs! This functionality is the premise of OpenMusic; you are using a program written in LISP to write other programs (your patches) in LISP. More information is easily found through a web search, for example at http://www.lisp.org/table/contents.htm

List

In LISP, a list is a group of elements enclosed in parentheses, for example (1 2 3). The elements may be of different types, for example (1 "hello" 34.5). A list may be empty: () or it may contain other lists as its elements: (1 2 (3 4) 5), in which case it is called a nested list or a tree. A list is a data type.

Loop

A loop is a portion of code which is repeated for every element of a set of data or as long as a certain condition is true, or both. OM provides this capability with a special function, `omloop`, though it is not the only way to create repeating code. Loops may contain other loops. This is called nesting; imagine a function which collected the names of all the children in a certain school, class by class. The function which asks for the child's name would be nested in a loop which executes as many times as there are kids in the class. This loop would *itself* be nested in another loop which repeated as many times as there are classes in the school.

Care must be taken when programming loops that the conditions for repetition are clearly specified and that the exit condition can be met at some point, otherwise the loop will continue to execute *ad infinitum*. This is especially important in recursive functions.

M

Maquette

A maquette is a special OM object incorporating a temporal dimension. A maquette has to axes, the x-axis representing elapsed time. Any playable OM object or patch can be placed within a maquette. Objects within a maquette always play based on their position along the x-axis, and also have access to data about their y-axis position and their shape, which you can then use to further influence their behavior. See the chapter on Maquettes for more information.

Macintosh Common Lisp

Macintosh Common Lisp, or MCL, is the standard for the LISP language on the Macintosh. Digitool MCL is a compiler for the LISP language, translating the LISP code of the OpenMusic software and of any functions you create within OpenMusic into machine code so that your computer can execute it..

Method

In a generic function, a method is a set of instructions that tell the function how to behave based on the data type(s) passed to the function when it is called. All functions have at least one method.

MIDI

An acronym for **M**usical **I**nstrument **D**igital **I**nterface, a standard of communication between digital music hardware agreed upon by all the major instrument manufacturers in the early 1970's. The MIDI standard contains 16 independant channels for communication; instruments can be set to receive on individual channels or on all channels simultaneously. There are 128 different types of messages that can be sent on the 16 channels, with some types being reserved for common information like Note-On (a key being pressed) or Note-Off (a key being released) and some types being left open for proprietary message types common only to instruments of a certain manufacturer. In order for OpenMusic to communicate with your MIDI instruments, you must have both a MIDI interface (a physical device connected to your computer translating the MIDI data into signals your computer can understand) and a MIDI driver or drivers (software programs that tell your machine how to talk to your MIDI interface). OpenMusic uses Midishare© to for input and output of MIDI data. You can configure Midishare© to send this data to your MIDI hardware or you can use Apple's built-in Quicktime© synthesizer to listen directly on your computer. See the Midishare documentation for more information.

Midic

An abbreviation of midicent. Midicents are OM's way of representing pitch. The system is based on the MIDI standard, where notes are represented by numbers. In MIDI, middle C is arbitrarily set to 60, so middle C♯ is 61, and so forth. OM divides each

of these MIDI semitones into 100 increments. Thus, middle C is represented by the midicent 6000. Midics have the advantage of being able to represent both quarter- and eighth-tone temperaments with whole numbers; middle C quarter-♯ is 6050, an eigth-tone below that would be 6025, etc.

N

Nesting

A programming concept, nesting refers simply to embedding one object or function or set of instructions within another. With functions, nesting is a powerful tool; nesting saves time by allowing the same function to be nested within several other parent functions, rather than re-writing the same code each time. The same function may be nested such that it is called repeatedly- this is called a loop. A function may even be nested within itself! This is an example of recursivity.

A second common use of nesting is within lists. A list, or series of lists, may be nested in another list. Imagine the roster of names of children in a school, sorted by class. Each name would be a list with two elements. Each class would be a list of names, and the master list would be a list of classes. It would be a list of lists of lists, i.e., a list nested three levels deep.

O

Object

An object is the unit of code in an object oriented programming language- this can be code for a function or for a data structure. It is an individually controllable element. Objects in OM include functions,factories (classes), folders,maquettes, and patches. They are represented in OM by graphic icons which you manipulate within the Workspace.

Object-oriented Programming

OM is an example of an object-oriented programming language. As its name suggests, the idea behind object-oriented programming is the manipulation of objects. Pieces of code for commonly used functions or data structures are considered to be discrete objects which are then called when needed by any function. This means that important code need only be written once since it can be called over an over again by any part of any program (including itself- see recursivity.) Objects can themselves be made up of other objects; working with an object-oriented programming language is sort of like building with an erector set.

P

Patch

Those of you who have used either very old analog synthesizers or the program Patchwork will recognize the term. The term itself comes from the early days of synthesizers, when sound-producing components were 'patched' together with cables to create a signal path. OM programming works in the same manner, and both the term and the concept have come back into vogue in other programs as well. A patch is a basic conceptual unit in OpenMusic. It is a user-defined object. Inside a patch you can connect functions and other objects, including other patches, together to perform musical tasks.

Pitch Class

All pitches representing a given frequency or its octave transpositions. This will include all notes of the same name as well as their enharmonic equivalents. All the Cs and B♯s on a keyboard are part of the same pitch class, for example. There are thus only 12 pitch classes, numbered 0-11.

Pitch Set

Simply, a group a pitch classes with no repetitions. A pitch set may be ordered, implying a horizontal or melodic dimension, or non-ordered, implying a vertical or harmonic dimension. Pitch classes are usually notated with the numbers 0-11 representing the 12 tones of the octave, enclosed in parentheses.

Predicate

Predicates are a special class of function with two distinguishing features: they only have one output; this output only produces one of two values, t or nil. Predicates perform some sort of test on their inputs and return t (true) if certain conditions are fulfilled. Otherwise, they return nil (false). Predicates include `om>`, `alldiff?`, and `omand`. They are usually used to control program flow by causing one task or another to be performed depending on whether certain conditions are fulfilled, or to exit or continue loops.

See Also: Boolean Logical Operators.

Prime Numbers

Prime numbers are numbers which can only be evenly divided by 1 and themselves. 1 is a special number, being itself 1. The fact that prime numbers have no integer divisors (except 1 and themselves) makes them useful for creating certain kinds of musical structures.

Programming Language

A programming language is any code which instructs the microprocessor of your computer to perform a task. In fact, your microprocessor only understands one type of code,

called Machine Code or Assembly Language, which is a small set of very basic instructions for the microprocessor which can be grouped to perform more complicated actions. A metaphor would be visualizing taking a sip of coffee as a series of small simple actions: "close fingers on mug, raise mug to mouth-level, bring mug to mouth, tilt mug," and so on. This was obviously very tedious, and it wasn't long after the advent of the microprocessor that programs were written to simplify this process by putting a layer of functionality between the programmer and the machine code. The programmers could than write more intuitive code like, "take a sip of coffee," which the program would then interpret and compile into machine code, breaking the sip of coffee down into its component actions. These programs, or **compilers**, were not mind readers, however. They demanded a certain syntax in order to be able to interpret. The syntax demanded by a particular compiler is known as a programming langauge. LISP is a programming language, the language in which OpenMusic is written. The compiler it uses is called **Digitool Macintosh Common Lisp**, or MCL.

R

Recursivity

Recursivity is a special property of the LISP language. When you use OM, you manipulate functions as graphic icons; recall that you can place a function icon anywhere, *including inside itself*. When a function contains a reference to itself, it is called a recursive function. Recursive functions constitute a special kind of loop. Since a recursive functions call themselves, it is important that they do this only on certain conditions-otherwise they will continue to call themselves on to infinity, eventually saturating the available memory and creating an error.

When might you use a recursive function? Recursive functions have many uses; a very common one is for performing some kind of task on a tree whose dimensions are unknown until the function is called. The tree to be processed is passed to the function, which processes each element sequentially until it detects that an element is itself a list. The function then calls itself, passing itself the sublist, and the newly called function processes the sublist in the same way. When the newly called function (and its child functions, if it called any) finishes, control passes back to the parent function.

S

Set Complex

A set complex is a group of pitch sets sharing a certain property. Examples would be auto-complementary and limited transposition.

Sieve of Eratosthenes

A method for finding *prime numbers* developed by the Greek mathematician Eratosthenes of Cyrene (approx. 275-195 B.C.E.). Eratosthenes was the first to accurately estimate the diameter of the earth. His method for finding prime numbers consists of starting with a set of all whole numbers, eliminating from that set the multiples of 2, then the multiples of the next number still available in the set of whole numbers, (in this case, 3) then the next number still available after that (here, 5, since 4 is a multiple of 2, already eliminated). In this way, the set of whole numbers is passed through a "sieve" which leaves only primes. An interesting web page can be found at http://www.math.utah.edu/~alfeld/Eratosthenes.html

slot

In a *class*, a slot is a space for holding data. A slot always contains a value of a particular *data-type*. A class can be defined as a group of slots and *methods* for processing data in those slots. In OM, slots are represented by the inputs and outputs of the *factory* icon.

T

Tree

In the LISP language, lists can contain any number or type of element; this includes other lists. When a list occurs inside another list, it is said to be nested. A nested list is also called a tree because each sublist can be visualized as a branching-out of the parent list. In this documentation, the term tree denotes a list which *may or may not* be nested. In contrast, the term list is used to refer to a list which is flat, i.e., none of its elements are lists.

V

Variable

A variable is a symbol which represents a quantity, as you'll rememeber from algebra. Computer languages use variables to perform calculations. Some variables have values which are permanent by general agreement, for example, the greek letter π represents pi, the ratio of the circumference of a circle to its radius. Most variables, however, have a value which is assigned to them at the moment they are declared. The graphic environment of OpenMusic spares you the trouble of having to deal with variables most of the time- the values are passed between functions by the patch cords you draw between them. The patch cord *is* the variable, in a sense. If you really need to, you can still set variables and read their values with the LISP functions `setf` and `eval`.

Velocity

Also called, "key velocity." In the MIDI standard, velocity is how dynamics are represented, being the velocity with which a key is struck. Velocity information is transmitted every time you play a note on a MIDI controller keyboard. Like many types of MIDI data, velocity is measured on a scale with 128 values, running from 0 to 127, inclusive.